

R Studio

de Rosario Ruiz
y Dan Contreras

A2CAPACITACION.COM

Índice

Primera edición

Curso de R y R Studio

Manual para el alumno

María del Rosario Ruiz Hernández



Curso de R y R Studio

De María del Rosario Ruiz Hernández

Guadalajara, Jalisco

Ontario Data Science Consultants Limited

Marzo 2021.   Primera Edición

Editor:

Dan Contreras

Diseño:

Samantha Cardenas Reus

Danely Legorreta Parra

Para más detalles ve a: https://a2capacitacion.com/

"Leyenda legal"

Tabla de contenido

Capítulo 1 | Instalando R

Capítulo 2 | Básicos para el uso de R

Capítulo 3 | Visualización con ggplot2

Capítulo 4 | Transformación y exploración de datos

Capítulo 5 | Importación de datos

Capítulo 6 | Datos ordenados

Capítulo 7 | Datos Relacionales

Capítulo 8 | Cadenas de Texto

Capítulo 9 | Manejo de fechas

Capítulo 17 | Redux con ggplot2

Apéndice A

Capítulo 1

Instalando R

Capítulo 1 | Instalando R

En este primer capítulo comenzaremos proporcionándote las herramientas necesarias para desarrollar tareas con **R**. Lo primero que necesitas es instalar el programa.

Para ello necesitaremos dos componentes; **R** y **RStudio.**

Para descargar **R** debes usar la siguiente liga: *(https://cloud.r-project.org/)*, aquí encontrarás la versión adecuada a tu equipo de trabajo, ya que existen versiones de este software para Windows, para MAC e incluso para Lunix. Una ves que ya hayas instalado **R**, deberás proseguir con la instalación de **RStudio.**

RStudio es un ambiente de desarrollo integrado para programar con **R**, lo cual es un ambiente sumamente intuitivo y amigable. Para descagar **RStudio** debes usar la siguiente liga *(https://rstudio.com/products/rstudio/download/).*

RStudio no se ejecutará a menos que hayas instalado previamente **R**.

R es un software de uso libre y trabaja con paqueterías o librerías. Una librería es una colección de funciones, datos y documentación sobre las funciones, que incrementan las potencialidades de **R**.

Las librerías en **R** deben instalarse solo una vez. Ya instaladas, es necesario decirle a **R** que deseamos utilizarlas en la sesión de trabajo en la que nos encontremos. Considera que siempre que salgas de **R**, la selección de librerías utilizadas, se desactivarán y en la nueva sesión será necesario volver a cargarlas (no instalarlas, recuerda que esto se hace una solo vez).

Para instalar una paquetería, por ejemplo, digamos **tidyverse**, basta con usar el siguiente comando;

```
install.packages(“tidyverse”)
```

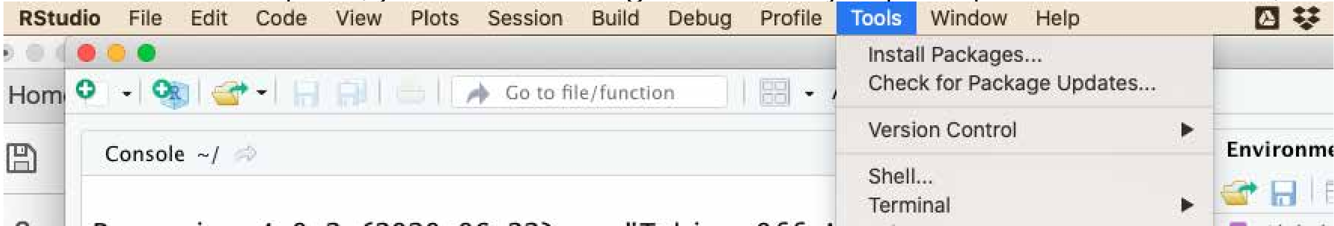

Mientras qué para activar su uso dentro de la sesión de trabajo es necesario el código;

```
library(tidyverse)
```

En ocasiones existen librerías que contienen dentro de sí mismas, mas librerías. Por ejemplo, **tidyverse** contiene dentro de si, las librerías; ggplot2, tibble, readr, dplyr, tidyr, purrr. Puedes instalar todo el conjunto de librerías presentes en **tidyverse**, o únicamente alguna de ellas. A lo largo de este curso iremos, usando diversas librerías. En cada caso te indicaremos cual paquetería debes instalar y activar para su uso.

Es recomendable mantener **R** y las librerías actualizadas. Para actualizar librerías puedes ejecutar **Rstudio** y usar el menú **Tools** o Herramientas, dependiendo el idioma que uses y seleccionar **Check for Package Updates.**

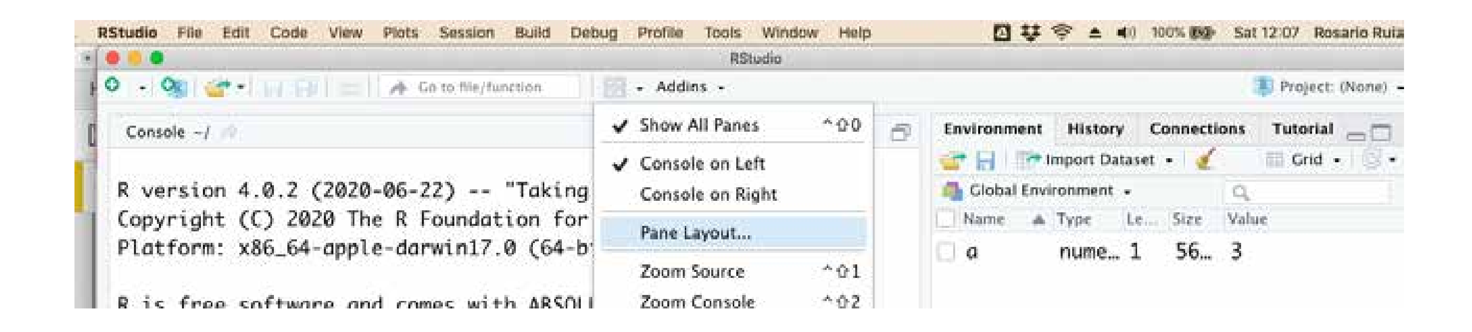
Durante el proceso de aprendizaje de **R** será común que te enfrentes a errores, los cuales siempre te aparecerán en color rojo, iniciando con la palabra **error**. Si tu **R** está en español, esos errores deberían estar en español, y si tu **R** esta en inglés los mensajes que **R** quiere darte a conocer, deberían estar en ese idioma. Si deseas cambiar de idioma los mensaje que **R** quiere darte durante la ejecución, considera alguna de estas dos opciones;




```
#Si deseas que sea en español
Sys.setenv(LANGUAGE = “en”)
#Si deseas que sea en inglés
Sys.setenv(LANGUAGE = “es”)
```

Con el fin de que te sientas cómodo trabajando con **R** es importante que utilices la configuración del entorno que prefieras. Para ello, una vez que ejecutas **RStudio** da click sobre la pequeña ventana que se encuentra en la barra de herramientas. En este espacio, ve a **Pane Layoult.**

Ahí podrás seleccionar el tipo y tamaño de letra, así la apariencia del entorno de trabajo.



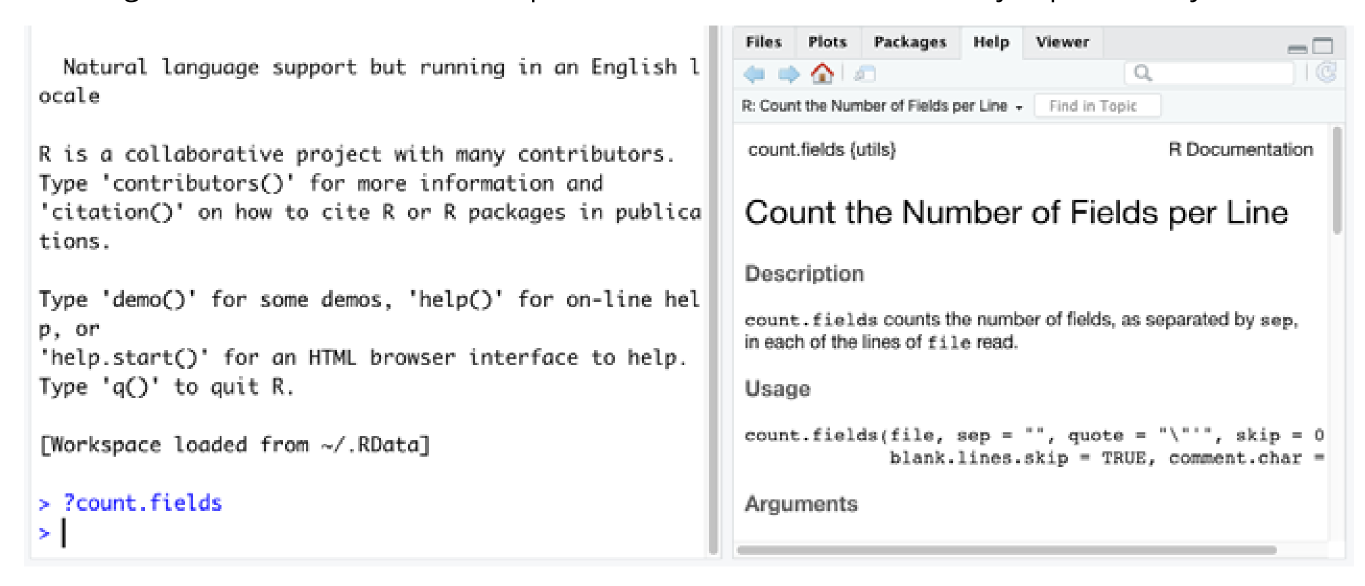
Este curso comienza con las ideas y expresiones básicas que debes tener sobre el manejo de datos en **R**. La importación, limpieza, transformación, análisis, visualización y presentación de datos, son el enfoque general del curso. En todo esto proceso la visualización y presentación son elementos centrales, pues de nada sirve todo lo anterior si no logramos comunicar los resultados. Al finalizar el curso, esperamos que por ti mismo seas capaz de construir documentos con un análisis sencillo, dentro de la funcionalidad de **R** conocida como **RMarkdown**, el cual constituye el objetivo final de este curso. Esta funcionalidad de **R** brinda ventajas importantes, para la presentación y visualización de información.

Si el proceso tiene dudas sobre el funcionamiento específico de una función, puedes pedirle a **R** que te de mas detalles sobre ella. Para ello puedes escribir en la consola el signo ? seguido del nombre de la función. Por ejemplo, prueba escribiendo lo siguiente en la consola de **R**

```
?count.fields
```

Pareciera que no se ejecuta ninguna indicación, sin embargo, del lado derecho en la pestaña **Help** o ayuda, se abrirá una ventana con la información de la función solicitada.

A lo largo del curso usaremos dos tipo de bases de datos; bases de ejemplos de **R** y bases con



datos reales de temas variados en México. El primero grupo de bases de datos se usará para ejemplificar procesos sencillos, el segundo lo usaremos para mostrar ideas y proceso más complejos.

Durante este proceso de aprendizaje será común que necesites mas ayuda de la que existe en este curso introductorio. Para ello necesitaras leer a detalle la documentación de las diversas funciones que **R** tiene. También puedes informarte sobre la evolución, mejoras y tips para el manejo de **R**, leyendo *(https://www.rbloggers.com/).*

Además, siempre puedes consultar *(https://stackoverflow.com/)* una plataforma pública que desde 2008 permite compartir código y conocimientos entre personas de todo el mundo. Puedes registrarte en la plataforma y hacer pregunta sobre los problemas que surjan mientras realizas código con **R**. Lo único que se te solicitará es que tus dudas estén acompañadas de un ejemplo replicable que el resto pueda hacer por su cuenta para poder ayudarte a resolver tus dudas.

Al hacer un ejemplo replicable, será necesario que incluyas una parte de la base de datos con la cual estás trabajando. Para que no tengas que cargar un archivo de la base de datos, puedes pedirle a **R** que te de el contenido de una base en formato de texto. Por ejemplo; en la consola escribe la siguiente línea

```
dput(mtcars[1:5,])
```

```
## structure(list(mpg = c(21, 21, 22.8, 21.4, 18.7), cyl = c(6,
## 6, 4, 6, 8), disp = c(160, 160, 108, 258, 360), hp = c(110, 110,
## 93, 110, 175), drat = c(3.9, 3.9, 3.85, 3.08, 3.15), wt = c(2.62,
## 2.875, 2.32, 3.215, 3.44), qsec = c(16.46, 17.02, 18.61, 19.44,
## 17.02), vs = c(0, 0, 1, 1, 0), am = c(1, 1, 1, 0, 0), gear = c(4,
## 4, 4, 3, 3), carb = c(4, 4, 1, 1, 2)), row.names = c("Mazda RX4",
## "Mazda RX4 Wag", "Datsun 710", "Hornet 4 Drive", "Hornet Sportabout"
## ), class = "data.frame")
```

Este comando trasforma todo el contenido de la base de datos en un formato de texto. Aquí el nombre de la base es **mtcars** y es una base predeterminada de **R**, hemos indicado la opción [1:5,] para indicar que deseamos de la fila 1 a la fila 5, de todas las columas (,) contenidas en esa base.

Copia toda la información que **R** te proporcionó y pegála con lo siguiente:

```
datos<-structure(list(mpg = c(21, 21, 22.8, 21.4, 18.7, 18.1, 14.3,
24.4, 22.8, 19.2), cyl = c(6, 6, 4, 6, 8, 6, 8, 4, 4, 6), disp = c(160,
160, 108, 258, 360, 225, 360, 146.7, 140.8, 167.6), hp = c(110,
110, 93, 110, 175, 105, 245, 62, 95, 123), drat = c(3.9, 3.9, 3.85,
3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92), wt = c(2.62, 2.875, 2.32,
```

```
3.215, 3.44, 3.46, 3.57, 3.19, 3.15, 3.44), qsec = c(16.46, 17.02,
18.61, 19.44, 17.02, 20.22, 15.84, 20, 22.9, 18.3), vs = c(0, 0, 1,
1, 0, 1, 0, 1, 1, 1), am = c(1, 1, 1, 0, 0, 0, 0, 0, 0, 0), gear = c(4, 4,
4, 3, 3, 3, 3, 4, 4, 4), carb = c(4, 4, 1, 1, 2, 1, 4, 2, 2, 4)),
row.names = c("Mazda RX4", "Mazda RX4 Wag", "Datsun 710",
"Hornet 4 Drive", "Hornet Sportabout", "Valiant", "Duster 360",
"Merc 240D", "Merc 230", "Merc 280"), class = "data.frame")
```

Esto permite construir una base de datos, sin la necesidad de incluir el archivo, ya que los datos se han trasformado en un formato de texto que puede ser compartido.

Con estos elementos, ya estamos listos para comenzar a trabajar.

Capítulo 2

Básicos para el uso de R

Capítulo 2 | Básicos para el uso de R

1 Introducción

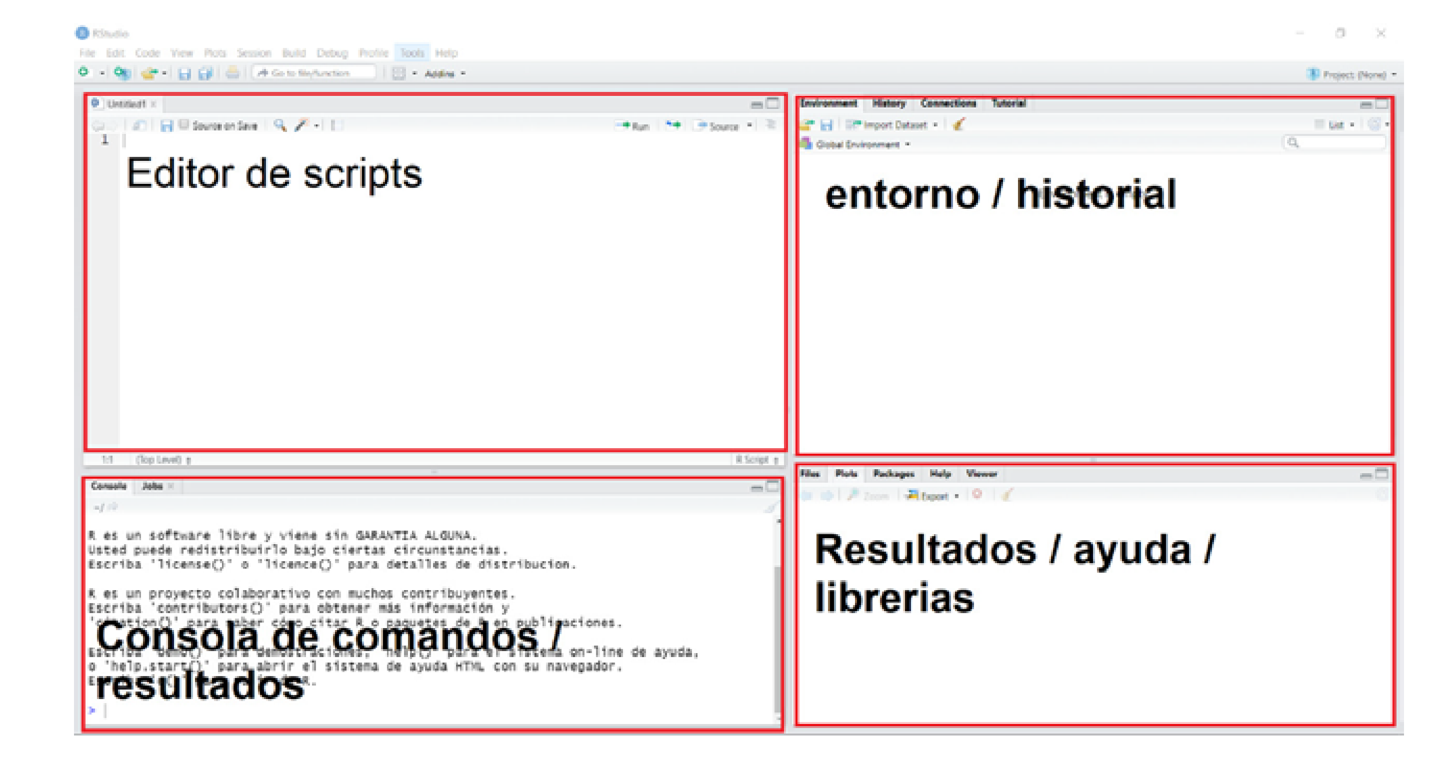
El objetivo de este apartado es conocer algunos detalles clave para el manejo de **R** y facilitar su eso al evitar errores comunes. **R** sigue una sintaxis que se debe respetar, es decir, un orden o reglas para escribir las instrucciones que permiten la correcta ejecución de tareas.

En este capítulo aprenderemos las funciones básicas de **R**; como asignar objetos, asignación de nombres y funciones. Conoceremos su entorno general y aprenderemos la importancia del uso de scripts para dar orden y facilitar la programación.

2 Interfaz de R

Una vez que hemos instalado **R** y antes de comenzar a realizar análisis y generar gráficas interesantes, es necesario que conozcamos la interfaz del software, cómo podemos personalizar la apariencia para sentirnos más cómodos mientras trabajamos y algunos consejos para ser eficientes con el uso de **R**.

Al abrir R lo primero que veremos es una ventana con cuatro cuadrantes, la cual luce mas o menos así:



• **Editor de scripts:** El cuadrante superior izquierdo es el editor de scripts, un script es un fichero de instrucciones, aquí podemos escribir todos los comandos que necesitemos para realizar nuestro análisis. La recomendación que te hacemos es que **siempre** trabajes con un script, si bien en algunas ocasiones podemos ejecutar algunas tareas directamente en la consola, es recomendable trabajar con un script. El código contenido en ellos se puede guardar, asegurando que el trabajo que desarrolles estará disponible para después.

- **Entorno / historial:** En este panel está compuesto por el entorno, donde se irán guardando los objetos con los que trabajemos y por el historial, en donde se registrarán todas las instrucciones ejecutadas.
- **Consola de comandos / resultados:** En la consola podemos escribir las instrucciones que deseamos realizan y el software las ejecutará inmediatamente, los resultados de las instrucciones que se ejecuten aparecen también en esta sección. El código que desarrolles en la consola no estará presente una vez que cierras **R.**
- **Gráficas / ayuda / librerías:** En el último cuadrante se encuentra un panel con diferentes opciones: explorador de archivos, gráficas, librerías y la ayuda.

Sí la primera vez que efectuaste **R** no observas los cuatro cuadrantes, no te preocupes. Del menú **Archivo** (o File si tu configuración es en inglés) selecciona **Nuevo Archivo/R Script**. Este generará el script, donde puedes empezar a escribir tu código.

2.1 Consejos

En medida de lo posible, para cada tarea que deseas realizar usando R te recomendamos usar un script, para que el uso de los scripts sea eficiente y no te pierdas entre tanto código te damos algunos consejos:

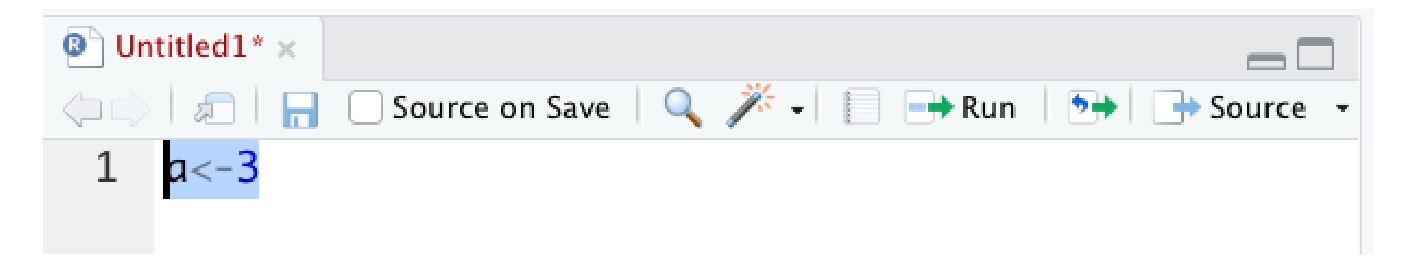
- Al guardar un script, se claro con los nombres, ser demasiado vago con los nombres o muy especifico puede ser una desventaja, por ejemplo, para este curso puedes hacer una script por capítulo, incluye en el nombre de archivo el número del capítulo, así si deseas regresar a un ejercicio en específico puedas encontrar rápidamente en que capítulo se encuentra. Igualmente, trata de no incluir espacios en los nombres de los scripts, puedes usar en su lugar _. **R** es un lenguaje construido originalmente en inglés, por lo que la recomendación incluye no escribir acentos en los nombres de los scripts, con el fin de evitar complicaciones.

- Se recomienda siembre establecer un directorio de trabajo, esto lo puedes hacer directamente en la pestaña **"session"**, **"set working directory"**, se recomienda elegir una carpeta donde estén las bases de datos que utilizaras en tu análisis, esto te permitirá evitar escribir toda la dirección del archivo. La primera vez que usemos esta opción dentro del curso, te enseñaremos a detalle como hacerlo. Esta indicación tambien se puede ejecutar usando la función **setwd()**, más adelante te daremos mas detalles de su uso.

- Trata de incluir una descripción de lo que estas haciendo dentro del script, así si después de un tiempo retomas algún proyecto puedas recordar fácilmente lo que hacías, también es útil mientras vamos aprendiendo a usar nuevas funciones. Además, si compartes o utilizas un script de alguien más es más fácil entender su funcionamiento. Para escribir texto en el script utiliza el símbolo **#** al iniciar una línea, así **R** entenderá que es una línea de texto y no una instrucción.

2.2 Ejecutar un script

Para echar andar nuestro análisis, podemos decirle a **R** que ejecute todo el script o solo una parte al seleccionar las líneas que deseamos ejecutar y utilizar el atajo **"ctrl + r"**. Este atajo dependerá del sistema operativo de ti computadora, y habrá diferencias entre si usas **iOS o Windows.** Si el atajo no funciona, usa la flecha verde **(Run)** en la parte superior del panel donde está el script. Los resultados del proceso de ejecución de un script se podrán observar ya sea en la consola, en el entorno o en el panel de resultados, dependiendo del proceso que se ejecute.



Si existe algún error de sintaxis en nuestro código, **R** nos dará una advertencia, colocando una cruz roja al inicio de la línea donde encontró el error, esto facilita la depuración del código.

3 Manejo de objetos con R

Para entender el funcionamiento básico de **R**, en este apartado vamos a desarrollar una serie de actividades sencillas que permiten familiarizarnos con la estructura de la sintaxis. En este caso y debido a que el código no es complejo, puedes escribirlo directamente en la consola o si lo prefieres, puedes crear tu primer script. Si decides crear el script no olvides las recomendaciones efectuadas y ejecutar el script para poder ver el resultado.

3.1 Objetos

R es un software integrado para manipulación de datos, hacer cálculos y crear visualizaciones. Literal, podemos usar **R** como una calculadora:

```
2+2
```

```
## [1] 4
```

```
pi*2
```

```
## [1] 6.283185
```

```
(4+3)/(3+0.7)
```

```
## [1] 1.891892
```

Puedes crear objetos usando el operador asignación "<-". Las entidades que utiliza **R** se llaman objetos, una entidad puede ser un valor, un número, una lista, un vector, una matriz, etc. Por ejemplo, en el capítulo anterior usamos el objeto **enoe**, el cual consistía en una base de datos contenida en un data frame.

En este ejemplo, el objeto "x" es el resultado de la operación **3*4**:

```
x <- 3*4
x
```

```
## [1] 12
```

Nota que al escribir únicamente el objeto x, **R** regresa el valor contenido.

Todas las instrucciones para crear objetos requieren asignar un argumento, la sintaxis es la siguiente:

nombre_objeto <- valor

De esta forma el objeto **nombre_objeto** es igual a **valor**. A lo largo de este curso se utilizará mucho el operador asignación, puedes usar el atajo **"Alt -"** para ahorrar algo de tiempo al escribirlo. Nota que al usar el atajo **"Alt -"** para escribir el operador asignación, el software agrega de manera automática un espacio antes y después del operador, se aconseja utilizar estos espacios, pues facilitan la lectura del código.

Una vez creado un objeto, podemos manipular su valor o contenido según nuestras necesidades, esto evita generar nuevos objetos. Además, también podemos reemplazar su valor utilizando otros objetos como lo muestra el siguiente ejemplo:

```
y <- 3*pi
y
```

```
## [1] 9.424778
```

```
y <- x*y
y
```

```
## [1] 113.0973
```

```
z <- x+y
z
```

```
## [1] 125.0973
```

Lo que observamos es la construcción de una primera variable de nombre **y** la cual guarda el valor de **tres veces pi**. Esa variable **y** ha sido modificada reemplazando nuevamente su valor por lo contenido, más lo que vale **x**. Finalmente en un objeto de nombre **z**, hemos almacenado la suma de **x** con **y**. Si en algún momento sientes que has creado demasiados objetos, puedes inspeccionar los objetos almacenados ejecutando la siguiente instrucción en la consola:

```
objects()
```

```
## [1] "x" "y" "z"
```

objects() te dará una lista de los objetos que están disponibles.

Para eliminar un objeto utilizamos la instrucción **remove()**:

```
remove(y, z)
```

Puedes asegurarte que los objetos han sido eliminados, usando nuevamente **objects()**.

3.2 Nombres

Los nombres de objetos pueden ser alfanuméricos, pero no comenzar con un número. Se recomienda que los nombres den una descripción breve de su contenido, puedes utilizar "_" para separar palabras y facilitar la lectura, por ejemplo:

```
base_triangulo <- 10
altura_triangulo <- 30
area_triangulo <- (base_triangulo*altura_triangulo)/2
```


Para inspeccionar el valor de un objeto, una vez creado, basta con escribir su nombre en la consola:

```
area_triangulo
```

```
## [1] 150
```

Una ventaja de **R** es que tiene integrado una función de auto-completado que te permite buscar funciones, argumentos y objetos. Basta con escribir un prefijo y el software nos mostrará los objetos con ese prefijo, de esta manera evitamos escribir nombres muy largos y errores de escritura.

```
este_es_ejemplo_de_nombre_largo <- 1
```

Para ver el valor de este objeto basta con escribir en la consola de comando "este" y después presionar la tecla **"Tab".**

El software también guarda un historial de las instrucciones que imputemos en la consola, si deseamos cambiar el valor del objeto "este_es_ejemplo_de_nombre_largo" con la fecha de navegación arriba podemos regresar a la instrucción anterior y volver asignar un nuevo valor diferente. Por lo cual, cada vez que presiones la tecla fecha hacia arriba, podrás explorar el historial de comandos ejecutados.

Los nombres importan, **R** es susceptible a mayúsculas y minúsculas y ortografía, es decir, los errores tipográficos son importantes. Debemos ser consistentes en los nombres que se asignen a los objetos.

Por ejemplo, si generamos un objeto y le asignamos la etiqueta **nombre** y mas tarde nos referimos a el, como **Nombre**, **R** nos mostrará un mensaje como el siguiente:

```
nombre <- pi
Nombre
```

3.3 Funciones

R utiliza varias funciones que se construyen con la siguiente sintaxis:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

Esta sintaxis indica que para usar este tipo de funciones es necesario escribir el nombre de la función y entre paréntesis los argumentos u objetos que requiere de entrada. En un ejemplo anterior utilizamos la función **remove()**, que utiliza este tipo de sintaxis.

Console | R Markdown ×

~/

```
> nombre <- pi
> Nombre
Error: object 'Nombre' not found
>
```

Probemos con otro ejemplo, la función **seq()**, la cual sirve para generar una secuencia de números.

En la consola escribe se seguido de la tecla **"Tab"**, la herramienta de auto-completar nos ofrece varias opciones, puedes añadir una **"q"** o utilizar las fechas de navegación para buscar la función. Observa que al buscar pasar sobre la lista de funciones aparece una descripción emergente que provee de información sobre la misma. **R** auto-completa también uso de paréntesis, comillas y otros signos.

```
seq            {base}    seq(...)
seq_along      {base}    Generate regular sequences. seq is a standard generic with a
seq_len        {base}    default method. seq.int is a primitive which can be much faster
seq.Date       {base}    but has a few restrictions. seq_along and seq_len are very fast
seq.default    {base}    primitives for two common cases.
seq.int        {base}    Press F1 for additional help
> seq
```

Utiliza la función seq(), para generar una secuencia de números de 1 a 10, con incrementos de uno en 1:

```
seq(1,10, by=1)
```

```
## [1]  1  2  3  4  5  6  7  8  9 10
```

En este caso la indicación **by=1** indica precisamente que el aumento es de 1, en 1.

Otra forma en la que el software nos brinda ayuda es alertando sobre errores en el uso de una función. Ejecuta esta instrucción en la consola:

```
hi ← "Hello
```

Al ejecutar esta instrucción notaras que en la consola aparece un signo "+", el cual indica que la instrucción esta incompleta y **R** espera que la termines. Puede ser por falta de una comilla o un paréntesis, en nuestro ejemplo, se debe a la falta de una comilla. Podemos añadir el elemento faltante y se terminará de ejecutar la instrucción o su en caso presionar **"Esc"** para cancelar la instrucción.

```
Console   R Markdown
~/
> hi<-"Hello
+
```


4 Directorios y Proyectos de RStudio

Hemos dicho que es importante siempre escribir nuestro código en un script, pues nos permitirá regresar a el más tarde. La pregunta que surge es ¿dónde se guarda mi script? La respuesta a ello es que tus scripts estarán en el directorio que hayas fijado con el uso de **setwd()**, también ahí se encontrarán todos los archivos de datos que exportes. Una vez que has fijado un directorio, con la indicación **setwd()**

Observarás en el encabezado de la consola, una ruta que indica que el directorio ha sido establecido.

```
Console   R Markdown
~/Dropbox/Curso de R/Cap2_Basicos_uso_R/
> setwd("~/Dropbox/Curso de R/Cap2_Basicos_uso_R")
>
```

Al fijar el directorio es necesario incluir una trayectoria completa y en este caso existen dos diferencias fundamentales, dependiendo del sistema operativo que se utilice. Por ejemplo; Mac y Linux usan / para definir las trayectorias, mientras que Windows usa ****. En lo que te familiarizas con la forma en tu equipo requiere indiques la trayectoria hacía un directorio, puedes usar **Fijar directorio** en el menú **session**. En el siguiente capítulo haremos uso de esta indicación para comenzar a trabajar con nuestra primer base de datos.

Si en algún momento tienes duda sobre cual es el directorio de trabajo, sobre el cual se esta guardando la información, puedes usar el comando;

```
getwd()
```

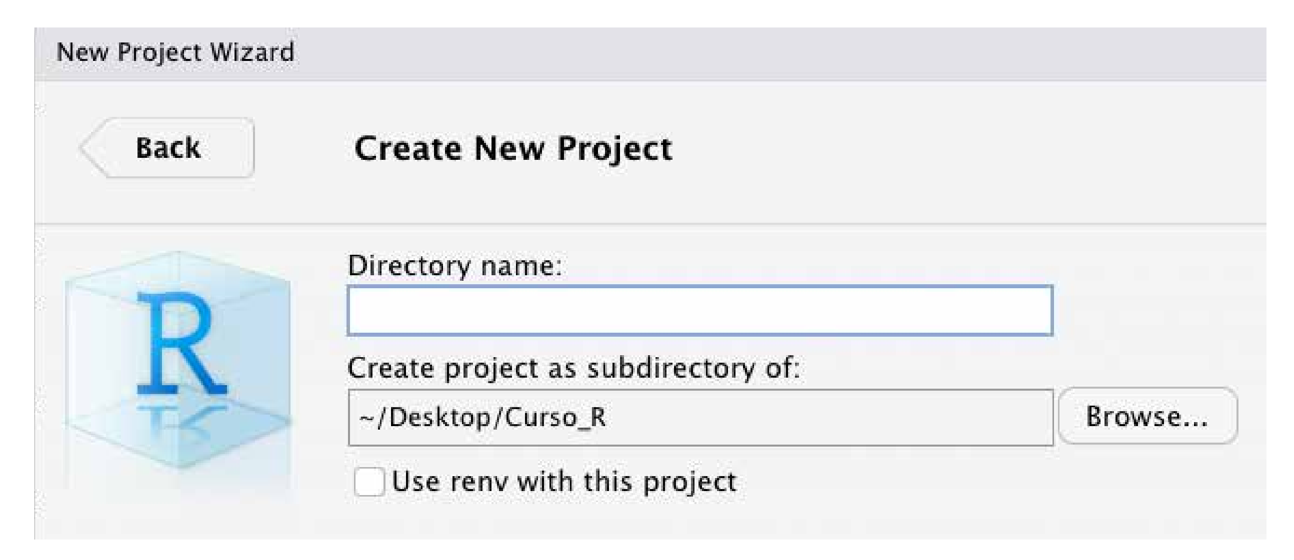
```
## [1] "/Users/RoseRuiz/Dropbox/Curso de R/Cap2_Basicos_uso_R"
```

Cada vez que cierres completamente **R**, este te preguntará si deseas guardar el **workspace**, esto guardara todos los objetos creados que se encuentren en el **enviroment**. La recomendación es que nunca guardes tu workspace, ya que el objetivo de toda buena estructura de programación es que las ejecuciones se puedan replicar usando únicamente el código, el cual debe estar contenido completamente en tu script.

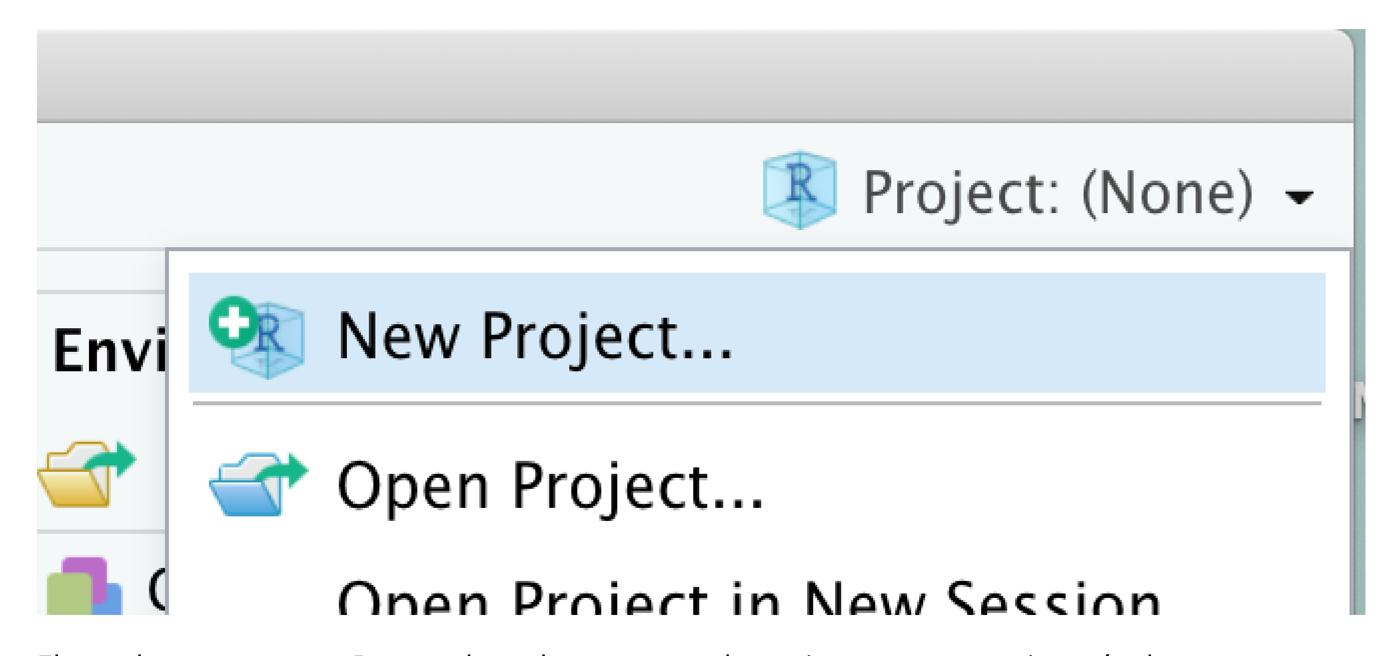
R permite crear proyectos donde de manera automática podemos agupar toda la información referente a un tema y cambiar entre proyectos de una forma sencilla. Esto es útil cuando se trabaja con diferentes proyectos a la vez. Para crear un proyecto basta con ir al menú archivo y seleccionar nuevo proyecto.



Después creamos un nuevo directorio, seleccionamos nuevo proyecto y definimos donde deseamos que se guarde la información. Esto nos permitirá tener ordenado nuestro trabajo.



Una vez que hemos creado uno o mas proyectos en R, podemos cambiar entre ellos en la barra de la parte superior derecha.



El uso de proyectos en **R** se vuelve relevante cuando se tienen que manejar más de un proyecto a la vez. De esta manera siempre que estés trabajando dentro de un proyecto y guardes cualquier cosa, podrás ir al directorio y ver que todo se encuentra en ese lugar.

5 Actividades

1. Genera los objetos necesarios para que z sea el resultado del área de un círculo.
2. Sí el radio del es 5, ¿cuál es el valor del área del circulo?
3. Genera una secuencia de números que tomen valores de 0.5 en 0.5 comenzando en 1 y terminando en 100.

Capítulo 3

Visualización con ggplot2

Capítulo 3 | Visualización con ggplot2

1 Introducción

Este capítulo se centra en la construcción de gráficos usando la paquetería **ggplot2**. Para ello utilizaremos la base de datos de la Encuesta Nacional de Ocupación y Empleo (ENOE), la cual busca medir el salario mensual (ingreso mensual) de los mexicanos y es elaborada por el Instituto Nacional de Información y Estadística (INEGI). Esta encuesta se realiza cada trimestre. Durante este capítulo analizaremos los salarios del cuarto trimestre del 2019. Con fines educativos, utilizaremos únicamente una muestra de esta encuesta.

Para trabajar con esta base de datos, es necesario primero cargarla a **R**, por lo que utilizaremos comandos básicos (o instrucciones) que respondan a este fin. Mas adelante, en el desarrollo el curso, ampliaremos con mayor detalle el proceso de importación de datos, el cual resulta tan amplio que dedicaremos un capítulo completo para ello. Durante el desarrollo de este capítulo, conoceremos un poco de los **data_frame** un tipo de objeto especial en **R** para almacenar información. Los detalles sobre el manejo de datos se abordan en los capítulos siguientes.

2 Previos

Antes de comenzar debemos de instalar las librerías necesarias para este capítulo, en el cual utilizaremos;

- ggplot2
- tidyr
- readxl
- dplyr

Una librería es un archivo donde previamente se encuentran un conjunto de comandos o instrucciones para desarrollar determinadas tareas.

Para instalar una librería en **R** se usa el siguiente comando

```
install.packages("ggplot2")
install.packages("readxl")
install.packages("tidyr")
install.packages("dplyr")
```

Una vez instalada debemos indicar a **R** que deseamos usarla, para ello basta con la instrucción.

```
library(ggplot2)
library(readxl)
library(tidyr)
library(dplyr)
```

Observa que para instalar la librería es necesario que su nombre se encuentre entre comillas, mientras qué para llamarla al entorno de R, no es necesario su uso.

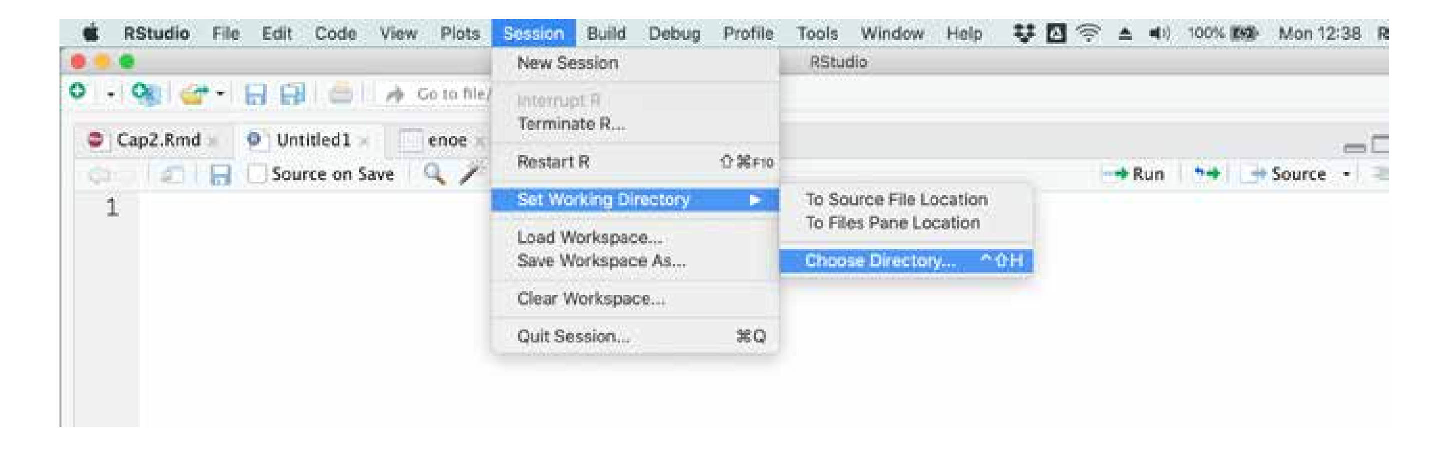
Esta instrucción habilitará la librería ggplot2 para ser usada durante la sesión de trabajo en R. Cada librería debe ser instalada una sola vez. Una vez instalada únicamente debemos decirle a R que deseamos usarla.

Durante tu proceso de aprendizaje, irás conociendo las paqueterías mas usadas y su funcionamiento. Por ahora podemos decir adelantar que **ggplot2** es la librería donde se encuentran los comando para las gráficas que efectuaremos, mientras que **readxl** contiene los comandos necesarios para poder cargar los datos que usaremos al entorno de **R**.

3 Cargar datos a R

La base de datos con la información requerida lleva por **nombre mu_enoe.xlsx**. La puedes obtener aquí. Descarga la base y guárdala en una ubicación en la que estés trabajando todos los documentos referentes a este **Curso de R**.

Para cargar la base de datos en **R** y poder trabajar con ella, debemos primero fijar un directorio de trabajo. Esto significa que le estamos diciendo a **R**, que de manera automática, busque archivos de bases de datos y demás, únicamente en la carpeta del directorio deseado. Esto es posible gracias al uso del comando **setwd**. Por ejemplo, en la computadora en la que estoy trabajando, existe una carpeta de nombre **Dropbox**, dentro de la cual se encuentra una carpeta de nombre **Curso de R**. En ésta última está contenido el archivo de Excel, sobre el cual tabajáremos. Si experimentas problemas fijando el directorio puedes usar el menú disponible en la parte superior de **R**.



Una vez que se fija el directorio, resta crear un objeto en el entorno de **R** que guarde la información contenida en nuestra base de datos. Debido a que la base de datos que estaremos trabajando se encuentra en formato de Excel, usaremos el comando **read_xlsx**, es decir leer excel. Debido a que ya hemos cambiado el directorio de trabajo, únicamente debemos escribir el nombre del archivo, el cual debe siempre ir entre comillas.

```
setwd("~/Dropbox/Curso de R")
enoe<-read_xlsx("mu_enoe.xlsx")
```

En la instrucción previa hemos indicado a **R** que cree un objeto de nombre enoe, la cual contendrá toda la información de la base de datos.

R nos permite cargar bases de datos en diferentes formatos, por ejemplo; podemos usar **read.csv**, para una base de datos que se encuentre en **csv**. Si este fuera el caso, deberíamos también instalar la librería **readr**. Estos elementos los profundizaremos con mas detalle en los capítulos siguientes.

4 Data Frame

La base de datos que hemos cargado a **R** y que esta contenida en el objeto **enoe** se conoce como un **data frame**. Podemos darnos cuenta de esto si usamos el comando class, el cual nos permite determinar el tipo de objeto en **R** con el que estamos trabajando.

```
class(enoe)
## [1] "tbl_df" "tbl" "data.frame"
```

Un data frame es un arreglo rectangular de variables (columnas) y observaciones (filas).

La base de datos enoe, contiene 12 variables y más de 8,000 observaciones. Cada observación corresponde a las respuestas que dio una persona a cada una de las 12 preguntas que se le hicieron. Usando el comando **dim** podemos descubrir el tamaño de la base de datos, y con el comando **colnames** los nombres de las variables que contiene.

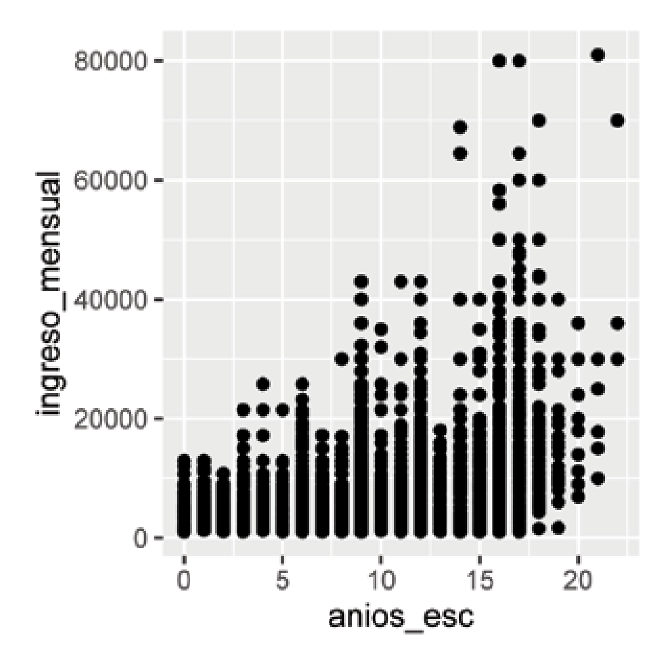

```
dim(enoe)
## [1] 10280 12
```

```
colnames(enoe)
## [1] "estado"   "sex"              "edad"           "asiste"
## [5] "pos_ocu"  "ing_salarios"     "niv_edu"        "anios_esc"
## [9] "hrsocup"       "ingreso_mensual"  "num_trabajos"   "tipo_empleo"
```

5 Uso de ggplot

Analicemos primero cual es la relación que existe entre los años de educación (variable anios_esc) y el ingreso mensual (variable ingreso_mensual). Para ello conviene efectuar un diagrama de dispersión o gráfica de puntos. La instrucción para generarla es;

```
ggplot(data = enoe) +
    geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual ))
```



Esta gráfica parece indicar que hay una relación entre el ingreso mensual y los años de escolaridad. Es posible apreciar que entre mas años de educación existan, el valor de la coordenada en y es más alto.

Siempre que desees crear una gráfica usando **ggplot**, debes comenzar la instrucción con el comando **ggplot**, el cual crea un sistema de coordenadas vacío. Este sistema se completa cuando se agrega la instrucción **geom_point**, la cual crea una capa de puntos sobre el sistema de coordenadas.

Con la indicación **data=enoe**, estamos indicando que en el objeto **enoe** se encuentran los datos de interés. La indicación **x =anios_esc** indica que deseamos observar los años de educación en el eje horizontal, mientras que **y =ingreso_mensual** indica que deseamos ver el salario o ingreso mensual en el eje vertical.

5.1 El formato básico

Una gráfica usando **ggplot2** tiene tres componentes básicos y la estructura más general de la instrucción se conforma por;

ggplot(data = **DATA**) + **GEOM_FUNCTION**(mapping = aes(**MAPPINGS**))

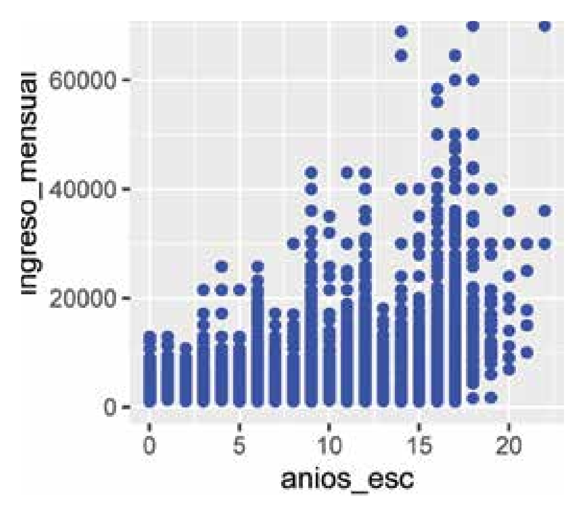
Donde:

- **DATA** se refiere al nombre del objeto donde esta almacenada la información
- **GEOM_FUNCTION** se refiere al tipo de gráfica a realizarse (por ejemplo; la función geométrica geom_point permite construir una gráfica de puntos, mientras que la función geom_bar permite construir una gráfica de barras)
- **MAPPINGS** se refiere a todos los componentes específicos de la gráfica que deseemos trabajar

En lo que resta de este capítulo aprenderemos diferentes **GEOM_FUNCTION** que nos permiten crear diferentes gráficas y una amplia variedad de **MAPPINGS**. Por ejemplo, si es de nuestro interés, podemos cambiar el color de todos los puntos, únicamente agregando la indicación color, dentro de las opciones de **MAPPINGS**



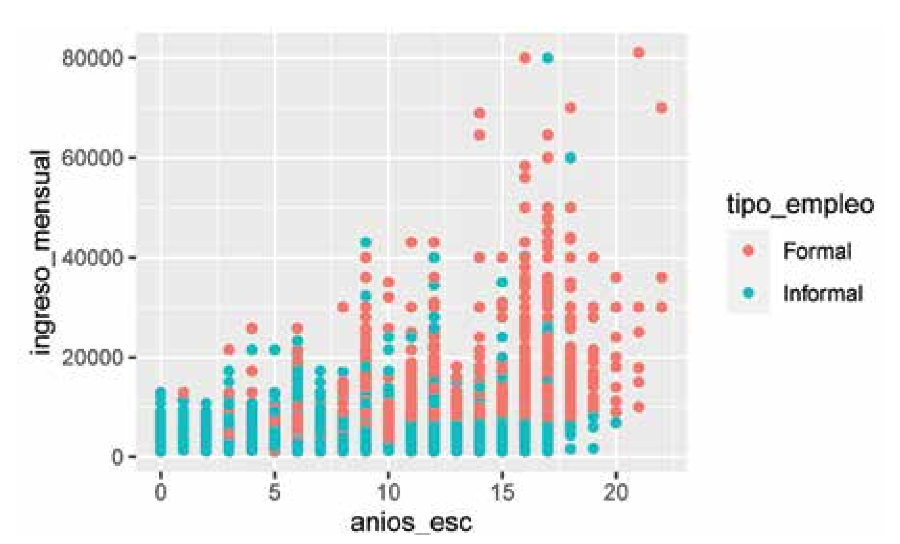
```
ggplot(data = enoe) +
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual ), color="blue")
```



5.2 Mapeos estéticos

Podemos profundizar un poco más en la visualización. En México muchas personas tienen un empleo con altos niveles de ingreso, aún cuando no tienen altos niveles de escolaridad. Esto es posible gracias a la existencia de un sector informal, del cual muchos mexicanos obtienen ingreso sin necesidad de tener un determinado nivel educativo. Para analizar cuales de las observaciones se trata de trabajadores del sector formal y cuales provienen del sector informal, podemos representar la misma gráfica, usando diferentes colores de puntos para cada uno de los sectores.

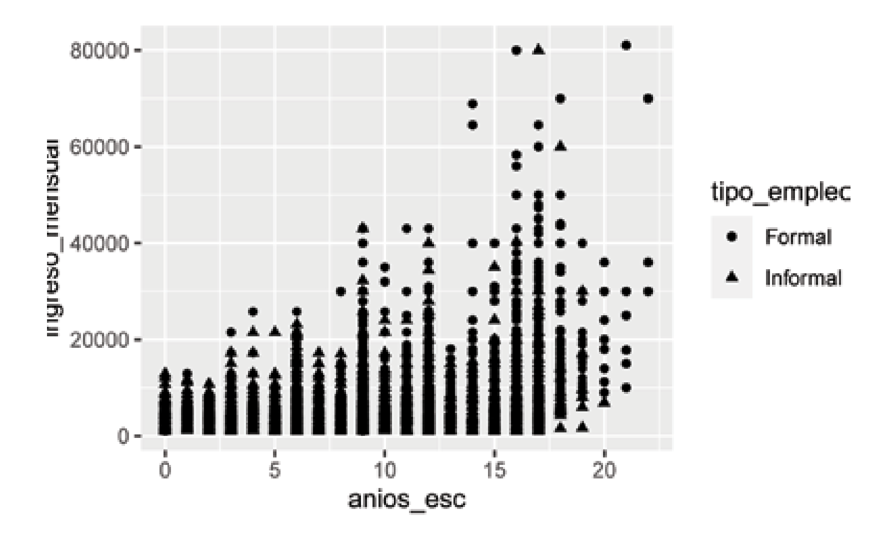
```
ggplot(data = enoe) +
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, color=tipo_em-
pleo ))
```



En esta nueva instrucción hemos pedido a **R** que diferencie cada observación según el **tipo_empleo**. Podemos hacer lo mismo pidiéndole que diferencie cada punto según el nivel educativo **niv_edu**, o cualquier otra variable. En este caso tenemos opciones de estética **aesthetics** que nos permiten observar una tercer propiedad de forma visual en los datos. Observa qué, en este caso, la indicación de color se encuentra dentro de la indicación **aes**. En general cualquier indicación que se encuentre fuera de la función **aes** modificará **todos** los puntos de la misma manera, mientras que si se encuentra dentro de la función **aes** cambiará la estética en relación al contenido de una variable. Esta variable debe ser indicada, como por ejemplo **color=tipo_empleo**

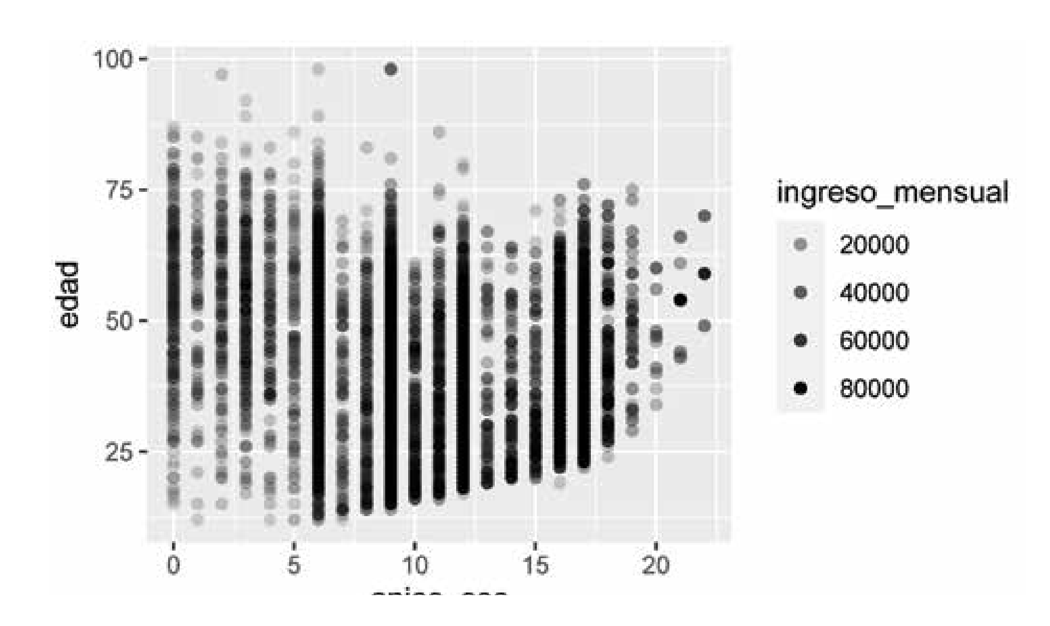
A cada **GEOM_FUNCTION** se le pueden asignar diferentes opciones de estética. Las básicosson; **color**, **size**, **alpha** y **shape**. Al igual que **color**, **shape** permite establecer una relación entre las categorías y los puntos que observamos en la gráfica. Para estos comandos debemos usar *variables categóricas*. En el caso de **size** y **alpha**, debemos usar *variables numéricas*. Ya que **size** generará un punto cuyo tamaño se relaciona con el valor de la observación, mientras que **alpha** generará una escala de transparencia dependiendo de la magnitud contenida en la observación.

```
ggplot(data = enoe) +
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, shape=tipo_em-
pleo ))
```



Esta gráfica nos muestra cada observación separada por tipo de empleo, usando un símbolo en lugar de usar un color.

```
ggplot(data = enoe) +
   geom_point(mapping = aes(x =anios_esc, y =edad, alpha=ingreso_mensual ))
```



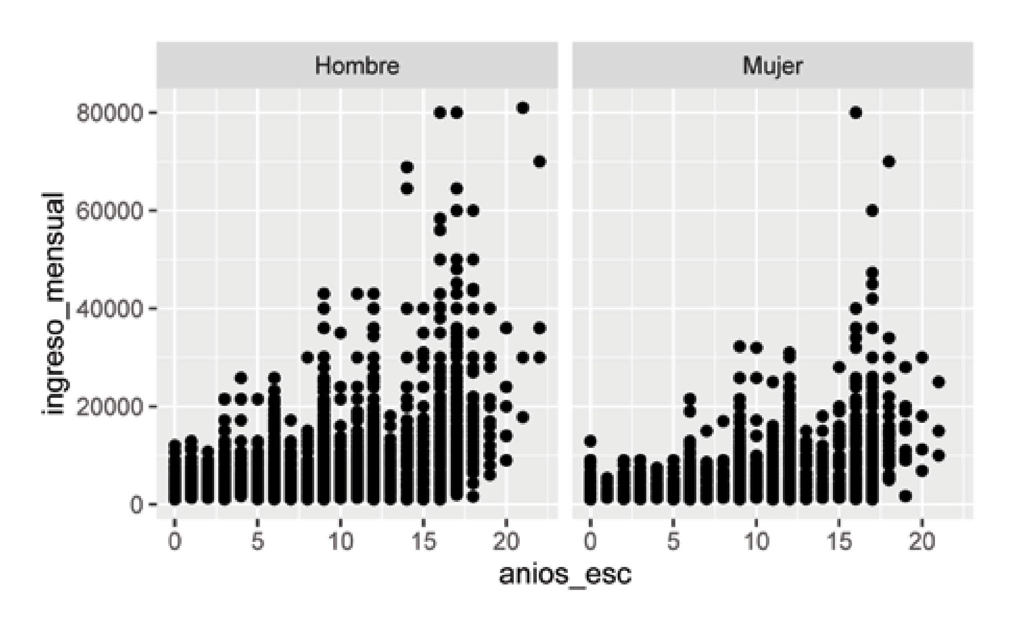
Al usar **alpha** podemos analizar la relación entre la edad, educación y el salario . Entre más obscuro es el color del punto, se trata de una observación con un mayor valor.

5.3 Separando en facetas

En la sección anterior, pudimos observar una tercera variable dentro de una gráfica bidimensional. **ggplot2** permite separar variables categóricas dentro de una misma gráfica, lo cual genera sub-graficas. Cada una de estas sub-graficas representa un subconjunto de los datos.

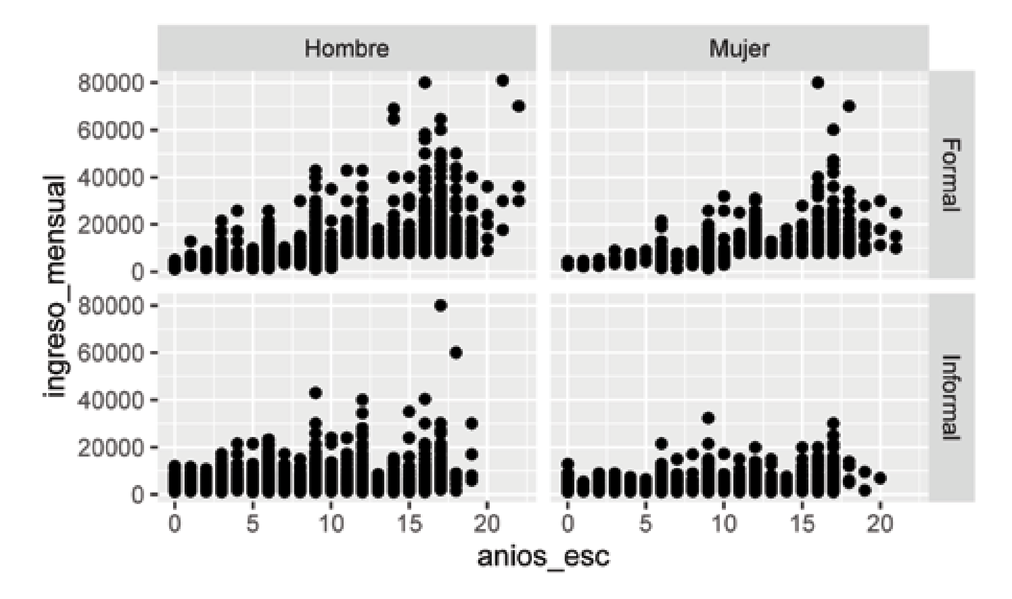
Este procedimiento es posible gracias al uso de la instrucción **facet_wrap.** El primer argumento de esta función debe ser una fórmula (en este contexto la palabra fórmula se refiere a una estructura de datos en **R**, no a una ecuación matemática). Una fórmula en R se expresa con el símbolo "~"1. Consideremos la instrucción.

```
ggplot(data=enoe)+
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))+
   facet_wrap(~sex, nrow=1, ncol=2)
```



En la gráfica mostrada hemos agregado la opción **facet_wrap** solicitando que separe la información por sexo **sex** indicando que la gráfica resultante la muestre en un panel que se compone de 2 columnas (ncol=2) y una fila (nrow=1). La información mostrada en la gráfica permite observar con mayor claridad las diferencias salariales entre los hombres y las mujeres, donde en términos generales los ingresos de los hombres son mas altos. Podemos crear un nuevo desagregado de información efectuando un cruce entre tipo de empelo y sexo. Para ello hacemos;

```
ggplot(data=enoe)+
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))+
   facet_grid(tipo_empleo~sex)[1]
```



Observa que en este caso hemos usado **facet_grid** y omitido la indicación sobre como deseamos que se distribuyan las sub-gráficas, por lo que **R** de forma automática, elige la distribución. La fórmula **tipo_empleo~sex** indica que deseamos que la información se desagregue tanto por tipo de empleo, como por sexo. Esta gráfica nos permite observar las diferencias en el salario de los hombres y las mujeres, tanto en el sector informal como en el sector formal, mostrando que estas diferencias son mayores en el sector formal.

Tanto **facet_wrap** como **face_grid** nos permite diferenciar de acuerdo con variables categóricas, por lo que debemos procurar incluir únicamente este tipo de variables.

5.4 Objetos geométricos

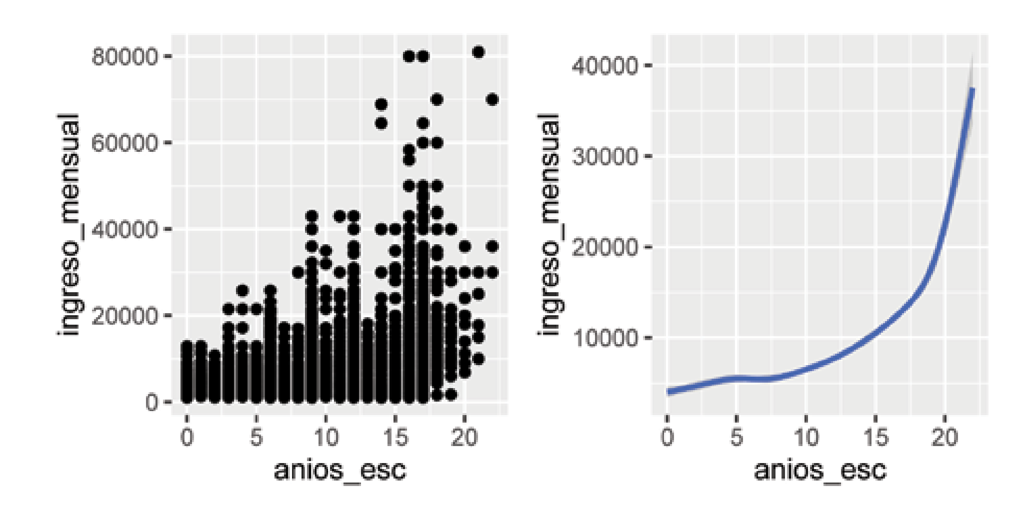
Anteriormente mencionamos que la instrucción básica para elaborar una gráfica usando **ggplot2**, consideraba ggplot(data = **DATA**) + **GEOM_FUNCTION**(mapping = aes(**MAPPINGS**))

Donde **GEOM_FUNCTION** hacia referencia al tipo de gráfica mediante el uso de diferentes **geoms**. En **ggplot2** un **geom** es un objeto geométrico usado para representar datos, cada representación distinta corresponde a un tipo diferente de gráfica. Para graficar barras usaremos **geom_bar** para gráficas de líneas geom_line. En ggplot2 es posible encontrar una amplia gama de posibilidades, pues proporciona mas de 40 **geoms**, los cuales puedes explorar con mas detalle aqui.

Para cambiar la geometría (tipo de gráfica), únicamente debemos cambiar el **geom_** que acompaña a **ggplot**.

Veamos los siguientes gráficas y las diferencias entre ellos

El gráfico de la izquierda corresponde a un gráfico de puntos (dispersión), resultante de usar **geom_point**, mientras que el de la derecha muestra una línea suavizada que ajusta al comportamiento de los datos. Los gráficos mostrados resultan de la ejecución del código, donde hemos incluido la opción **geom_smooth**.

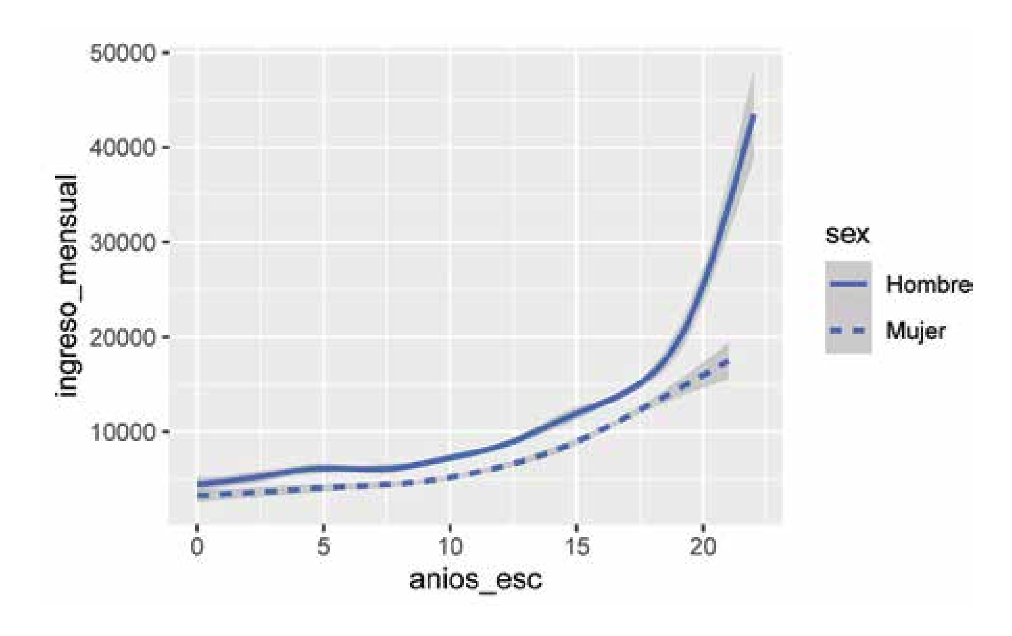



```
# Izquierda
ggplot(data=enoe)+
     geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))
# Derecha
ggplot(data=enoe)+
     geom_smooth(mapping = aes(x =anios_esc, y =ingreso_mensual))
```

No todos los elementos de estéticas de los que hemos hablado con anterioridad **(aesthetics)** se pueden aplicar a todas las geometrías **(geoms)**. En el caso de **geom_smooth** también es posible diferenciar entre distintas variables categóricas, para ello podemos usar **linetype**, esto generará un ajuste aproximado al comportamiento de los datos, haciendo diferencias entre los grupos que la conforman, por ejemplo;

```
ggplot(data=enoe)+
     geom_smooth(mapping = aes(x =anios_esc, y =ingreso_mensual, line-
type=sex))
```

La gráfica muestra dos líneas de ajuste del comportamiento de los datos, una para cada tipo de

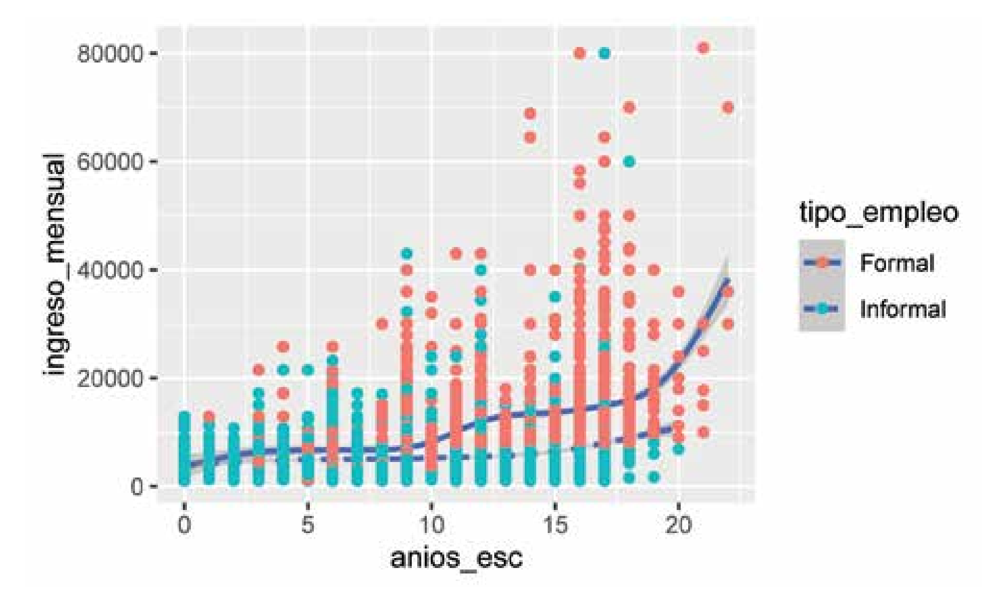


observación según la variable **linetype=sex**. Esto nos permite observar que para un mismo número de años de escolaridad, en promedio se espera que las mujeres ganen menos que los hombres.

Sobre esta misma gráfica es posible construir la gráfica de puntos o de dispersión que nos permiten mostrar cada una de las observaciones. Para ello lo único que debemos hacer es combinar dos geometrías en una misma gráfica. Esto, además de mostrar el comportamiento de ajuste en los datos, muestra cada una de las observaciones en el color con relación al tipo de empleo.

```
ggplot(data=enoe)+
     geom_smooth(mapping = aes(x =anios_esc, y =ingreso_mensual,
     linetype=tipo_   empleo))+
     geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual,
     color=tipo_empleo))
```

En general con **ggplot2** podemos construir gráficas con más de una geometría. Si observamos



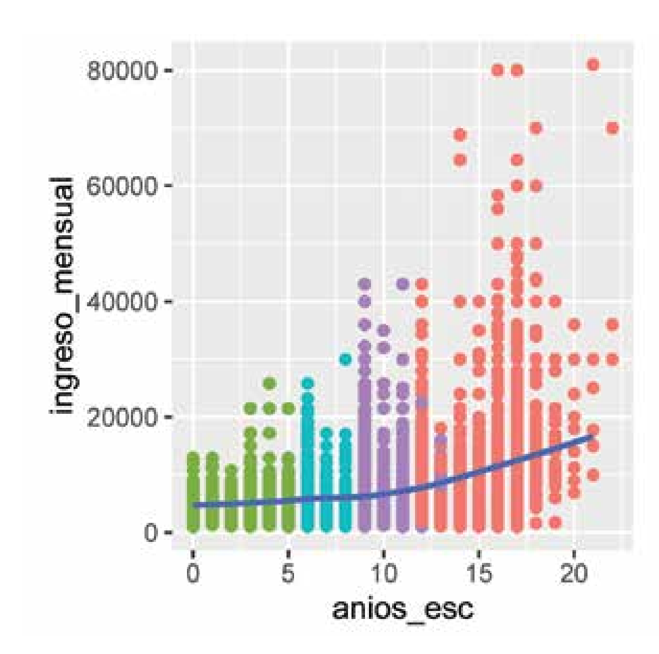
detalladamente el código que genera la gráfica anterior implica cierta duplicidad, pues tenemos que indicar en ambas geometrías, las variables que deseamos graficar. Para evitar esa duplicidad, podemos incluir de manera genérica dentro de la función **ggplot** las variables a gráficar y únicamente incluir los **geoms** deseados con sus correspondientes **mappings**. De manera que la gráfica anterior tambien puede ser construida con el siguiente código, el cual puedes ejecutar para comprobar que obtenemos el mismo resultado.

```
ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_smooth(mapping = aes(linetype=tipo_empleo))+
     geom_point(mapping = aes(color=tipo_empleo))
```

Al momento de tener una cantidad importante de gráficas, estos pequeños cambios se vuelven importantes y simplifican el trabajo y disminuyen la posibilidad de errores.

Usar varias geometrías en una misma gráfica, también nos da la posibilidad de usar diferentes conjuntos de datos para cada una de las capas en la gráfica. Por ejemplo, podríamos desear ver una gráfica de puntos que incluya todos los datos y una linea de ajuste a los datos, que represente únicamente el caso de Jalisco.

```
ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_point(mapping = aes(color=niv_edu), show.legend = FALSE)+
     geom_smooth(
         data=filter(enoe, estado=="Jalisco"), se=FALSE)
```

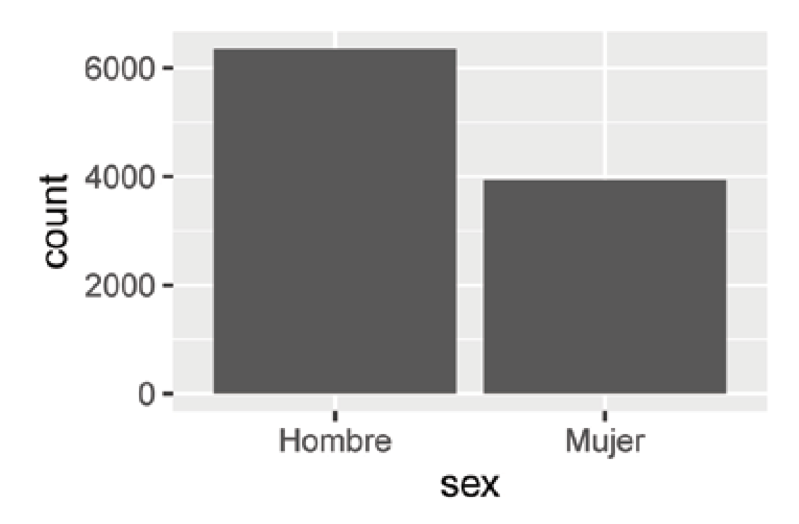


En esta instrucción hemos usado la función **filter**. En los capítulos siguientes profundizaremos sobre su uso. Hemos agregado también la opción **show.legend** la cual nos permite omitir de la gráfica la leyenda con la codificación del color de los puntos.

5.5 Transformaciones estadísticas

La base de datos de la enoe con la cual estamos trabajando contiene mas de 8,000 observaciones. Una forma fácil de ver cuantas de esas observaciones corresponden a hombre y cuales, a mujeres, es con una gráfica de barras.

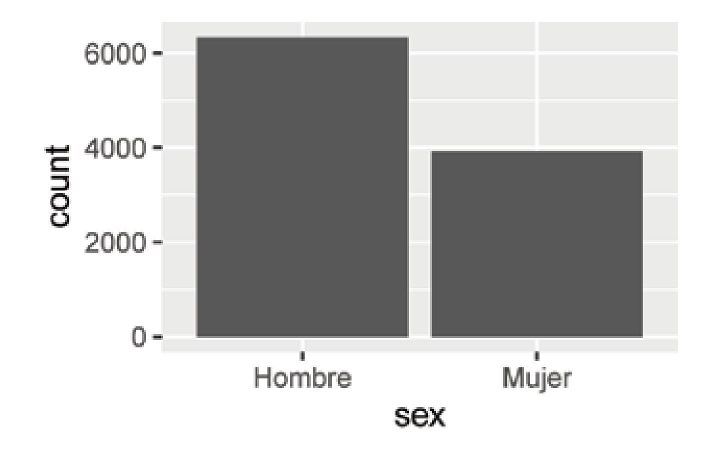
```
ggplot(data=enoe)+
     geom_bar(mapping = aes(x=sex))
```



En este caso la indicación permite contabilizar cuantos datos de cada categoría existen dentro de la base de datos. La categoría (en este caso hombre o mujer), se muestra en el eje x, mientras que la cantidad de observaciones de cada categoría se muestra en el eje y. La información que se observa en el eje y (altura de la barra), no se encuentra de manera directa en la base de datos (es decir, no existe una variable de nombre **count)**, y la instrucción que hemos dado obliga necesariamente a hacer un conteo para cada categoría, el cual se generó de manera automática. El algoritmo que se ejecuta para esto se conoce como **stat** (transformación estadística).

Cada geometría realiza una transformación estadística específica y cargada de manera predeterminada, en el caso de **geom_bar** se efectúa un conteo, el cual también puede ser ejecutado con la indicación **stat_count**. Generalmente las **geoms** y las **stas** son intercambiables. Por ejemplo, la gráfica de barras anterior puede ser creada usando la indicación

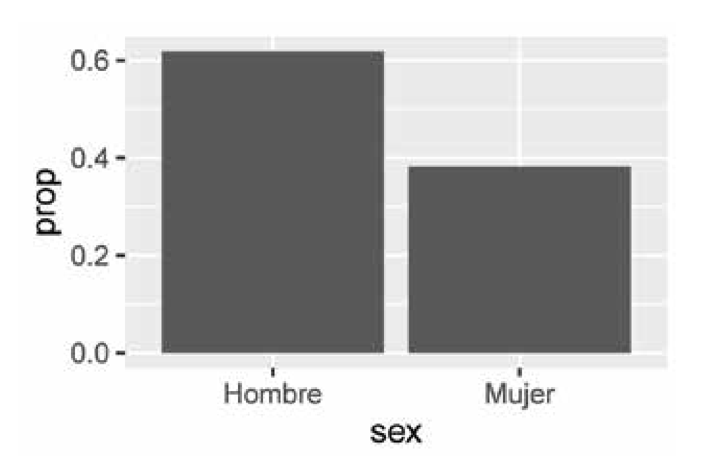
```
ggplot(data=enoe)+
     stat_count(mapping = aes(x=sex))
```



Para conocer detalladamente cual es la transformación estadística asociada a cada **geom**, escribimos **?**, acompañado del nombre del geometría, **?geom_bar**. Esto nos mostrara toda la documentación que existe referente a **geom_bar**. En **R** podemos usar este mismo procedimiento para conocer con mayor detalle el funcionamiento de cualquier comando.

También es posible expresar la cantidad de observaciones por categoría como una proporción. Para ello tendríamos que indicar lo siguiente;

```
ggplot(data=enoe)+
     geom_bar(mapping = aes(x=sex, y=..prop.., group=1))
```



En este caso mas que saber cuantos hombre y cuantas mujeres hay en la muestra, sabemos que mas del 60% de observaciones corresponden a hombres. Esto gracias a que hemos indicado que en el eje y deseamos observar la proporción **y=..prop...**

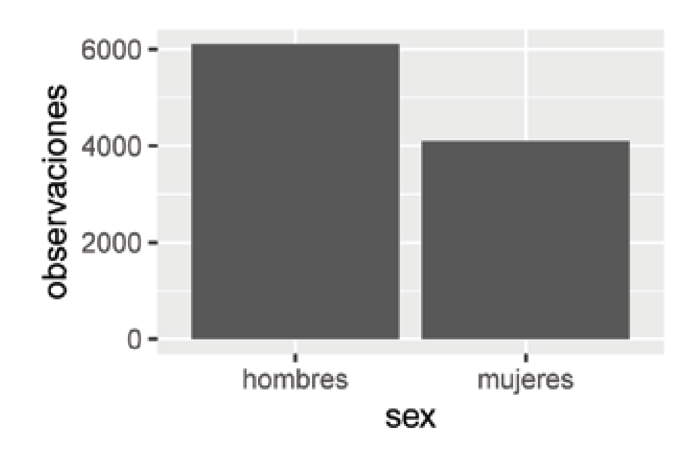
Recuerda que cada geometría tiene por defecto una transformación estadística asociada.

Si fuera el caso en el que la base de datos que estamos trabajando contiene además de las variables, el valor agregado de cada una de ellas, por ejemplo que en la base de datos tuviéramos una variables que totaliza el numero de hombres y de mujeres, al efectuar la gráfica de barras tendríamos que definir la transformación estadística como **identity,** que en realidad indica que no deseamos que se lleve a cabo ninguna transformación, ya que en ese caso se supone que el conteo ya esta hecho. Para ejemplificar este caso, supongamos que tenemos una base de datos que ya contiene la siguiente información;

```
##   sex       observaciones
## 1 hombres        6100
## 2 mujeres        4100
```

En este caso, ya conocemos el total de elementos en cada categoría, por lo cual la gráfica de barras puede ser ejecutada usando;

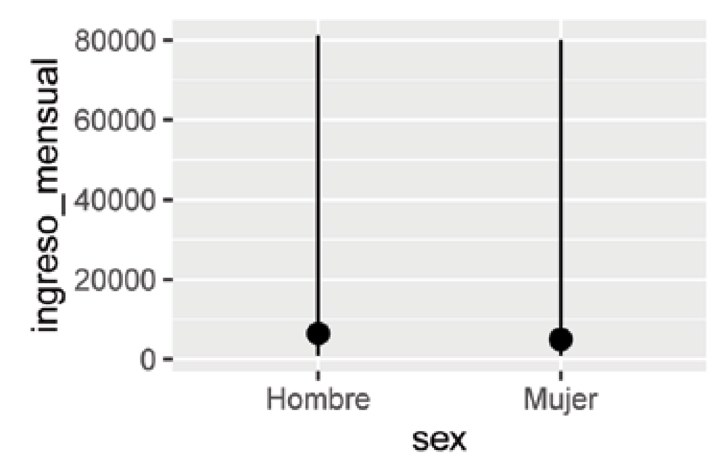
```
ggplot(data=prueba)+
     geom_bar(mapping = aes(x=sex, y=observaciones), stat="identity")
```



En este caso, ya no necesitamos hacer un conteo por cada categoría, debido a que la base de datos ya la tiene. Observa que ahora hemos declarado dos variables tanto **x=sex**, como **y=observaciones**, pues ambas variables existen dentro de la base de datos.

Regresando a nuestra base de ejemplo **enoe**, supongamos que deseamos representar el promedio del ingreso de cada grupo, así como los ingresos máximos y mínimos para cada grupo. En este caso podemos usar la transformación estadística **stat_summary**, la cual muestra un resumen de los estadísticos principales de una variable. Para ello, tendríamos que indicar **

```
ggplot(data=enoe)+
    stat_summary(
     mapping = aes(x=sex, y=ingreso_mensual),
     fun.min=min,
     fun.max=max,
     fun=median
    )
```



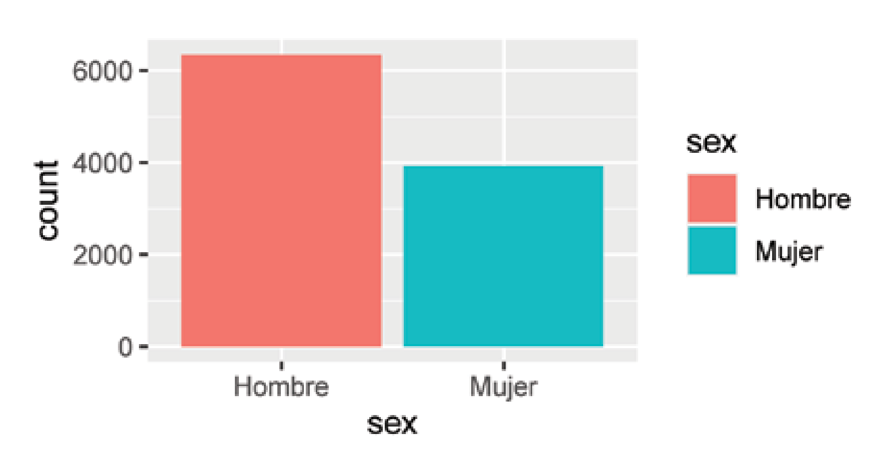
La indicación **fun=median** indica una transformación en la que es necesario calcular el promedio de la variable **y=ingreso_mensual** sobre cada una de las categorías.La gráfica nos permite observar que los ingresos mínimos y máximos entre hombres y mujeres son iguales, con una ligera diferencia en la mediana.

5.6 Ajustes de posición

5.6.1 position="stack"

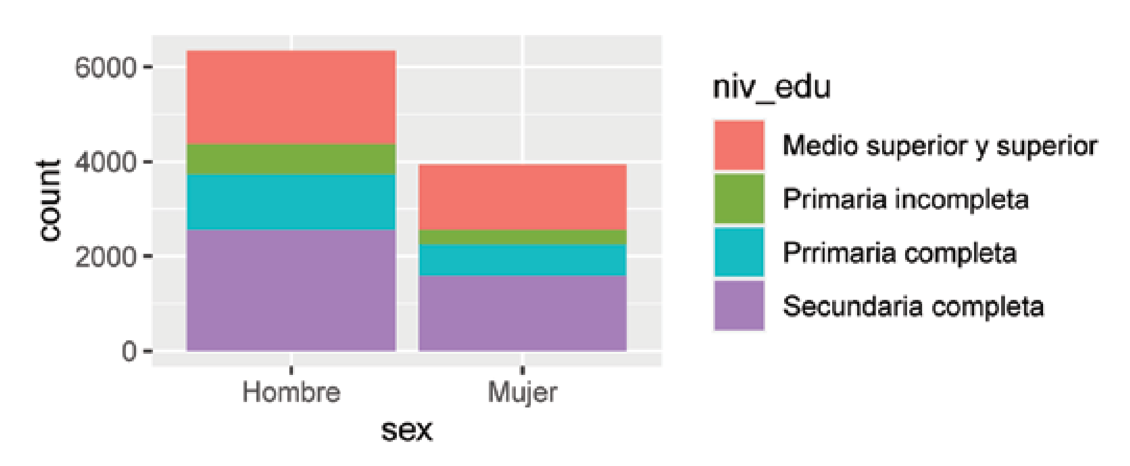
Con **ggplot2** es posible asociar un color a cada una de las diferentes categorías en las barras, para ello usamos la estética **fill.**

```
ggplot(data=enoe)+
     geom_bar(mapping = aes(x=sex, fill=sex))
```



En este caso hemos indicado la misma variable, tanto para el conteo de la categoría como para el color. Podríamos usar una variable distinta para observar el desglose de hombres y mujeres separando por sus diferentes niveles educativos. Para ello hacemos;

```
ggplot(data=enoe)+
     geom_bar(mapping = aes(x=sex, fill=niv_edu))
```



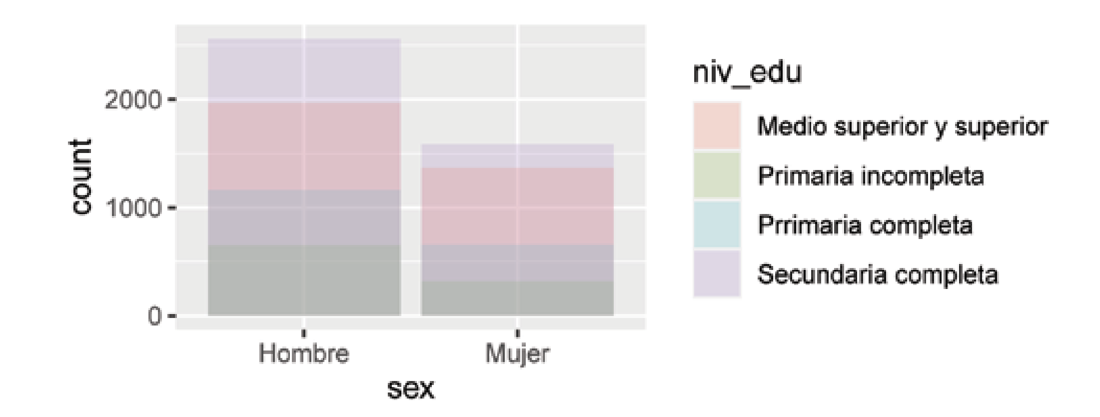
De forma automática se han apilado las diferentes observaciones de nivel educativo sobre cada una de las categorías. Esto significa que podemos observar cuantos hombre o mujeres dentro de la muestra, tienen determinado nivel educativo. En la indicación de manera predeterminada se incluye un ajuste en la posición **position="stack"**, para corroborarlo, puedes ejecutar el siguiente código y descubrir que se obtiene exactamente la misma gráfica. - *ggplot(data=enoe)+ geom_bar(-mapping = aes(x=sex, fill=niv_edu), position = "stack")*

Si no deseamos que la gráfica este apilada, existe tres posiciones más que podemos usar; **"identity", "dodge"** o **"fill"**.

5.6.2 position="identity"

Analiza las siguientes indicaciones

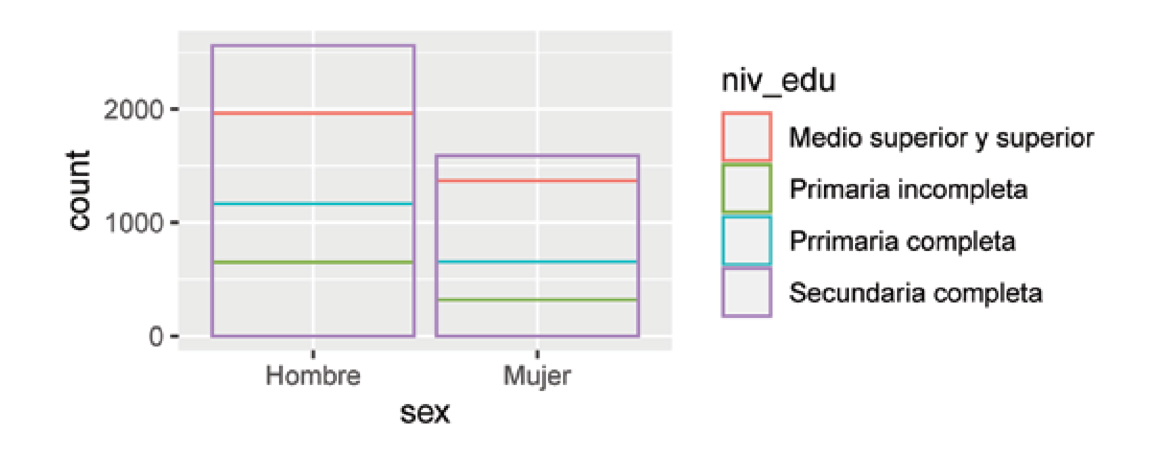
```
ggplot(data=enoe, mapping = aes(x=sex, fill=niv_edu))+
     geom_bar(alpha=1/5, position = "identity")
```



En este caso **position="identity"** indica el total de hombre y mujeres por cada nivel educativo, con la diferencia que las barras no están apiladas (puedes notar esto observando los números del eje y). El inconveniente de su uso es que las barras se traslapan, por ello es que para observarlas hemos incluido la indicación **aplha=1/5** la cual como vimos anteriormente indica transparencia y aplica para todos los casos pues se encuentra fuera de las opciones aes().

Otro ejemplo del uso de **position="identity"** en el cual se puede observar de mejor manera el traslape de las gráficas es la siguiente.

```
ggplot(data=enoe, mapping = aes(x=sex, color=niv_edu))+
     geom_bar(fill=NA, position = "identity")
```

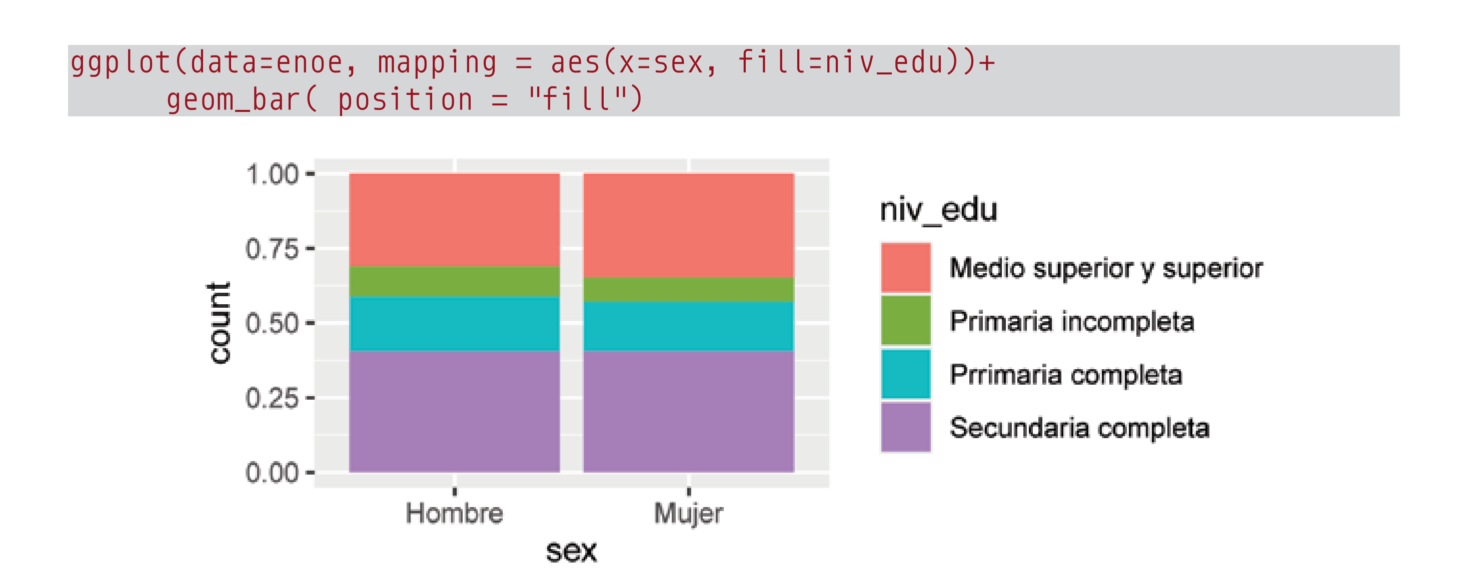


En este caso hemos incluido *fill=NA* indicando no deseamos ningún color de relleno de las barras. Nuevamente como se encuentra fuera de las opciones de aes() se aplicará para todas las categorías de la gráfica.

5.6.3 position="fill"

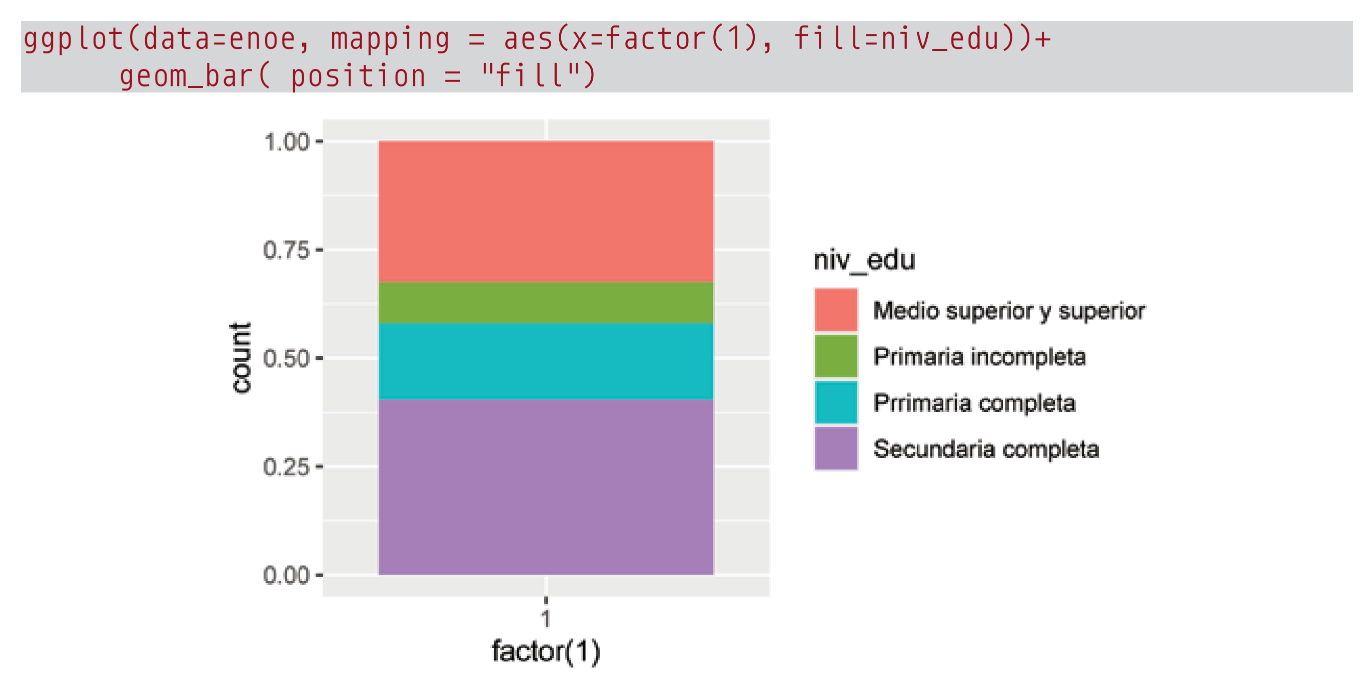
La indicación position="fill" funciona de manera semejante que stacked con la diferencia que todas las barras tienen la misma altura y cada una presenta el 100% de la categoría, lo cual permite observar con mayor claridad la proporción de cada clasificación dentro de la categoría.

```
ggplot(data=enoe, mapping = aes(x=sex, fill=niv_edu))+
     geom_bar( position = "fill")
```



Supongamos ahora que queremos ver en una sola barra únicamente con la distribución por nivel educativo. En este caso la indicación será

```
ggplot(data=enoe, mapping = aes(x=factor(1), fill=niv_edu))+
     geom_bar( position = "fill")
```



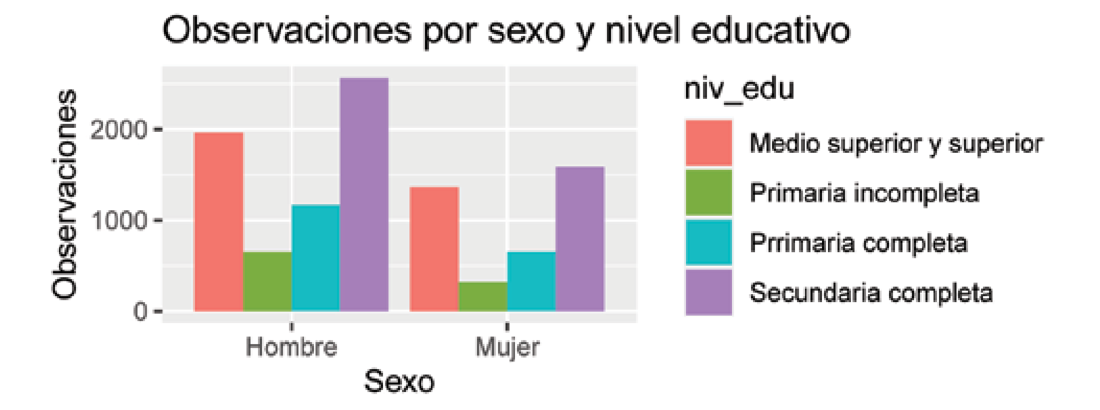
Observa que hemos usado **x=factor(1)** pues únicamente queremos que en un solo grupo se reúnan los diferentes valores de la variables contenida en **fill**.

5.6.4 position="dodge"

Finalmente **position="dodge"** no apila, ni traslapa las barras, mas bien las agrupa una al lado de la otra.

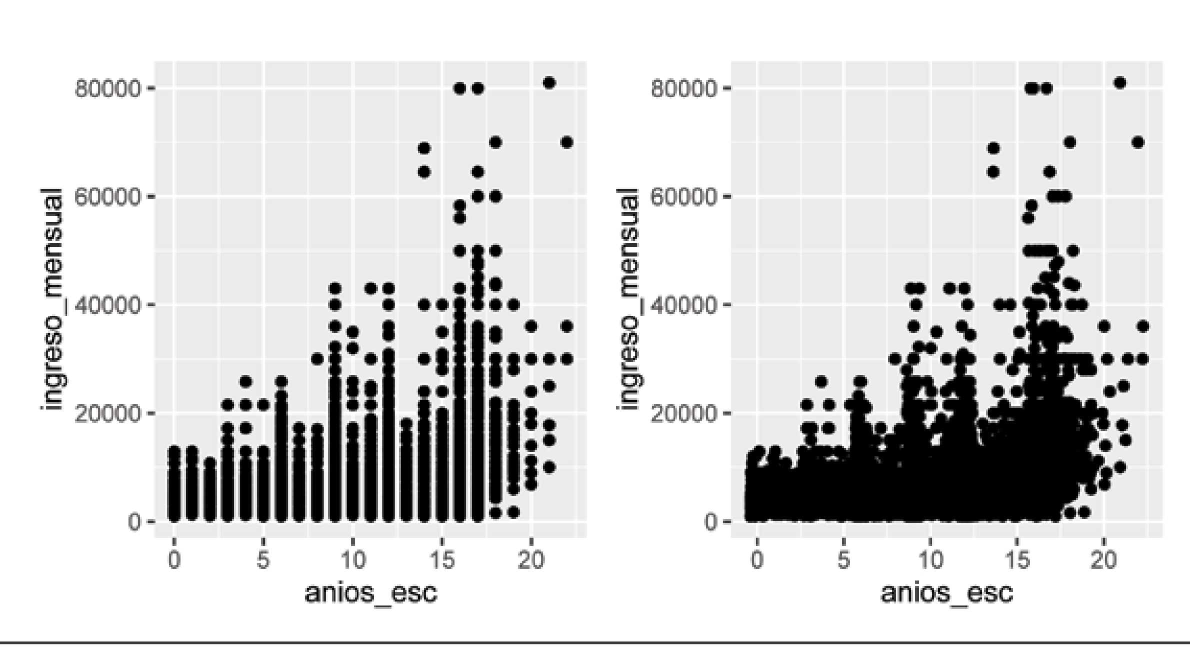
```
ggplot(data=enoe, mapping = aes(x=sex, fill=niv_edu))+
       geom_bar( position = "dodge")+
       labs(title="Observaciones por sexo y nivel educativo", x="Sexo", y="-
Observaciones")
```

Observa que hemos añadido un título a la gráfica y hemos cambiado el nombre de los ejes usando **labs()**, en cuyo profundizaremos más adelante.



5.6.5 position="jitter"

Para entender el funcionamiento de **position="jitter"**, observemos las siguientes dos gráficas;



La gráfica de la izquierda que muestra muchos de los puntos estan traslapados unos con otros. Esto se conoce como **overplotting**, lo cual complica la visualización de los datos. La indicación **position="jitter"** agrega un pequeño valor aleatorio a cada punto, para evitar que se sobrepongan unos con otros. Esto permite observar con mayor claridad la distribución de las observaciones. El código utilizado para generar estás graficas es;

```
#Izquierda
ggplot(data = enoe) +
     geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual ))

#Derecha
ggplot(data = enoe) +
     geom_point(mapping = aes(x=anios_esc, y =ingreso_mensual ), position =
"jitter")
```

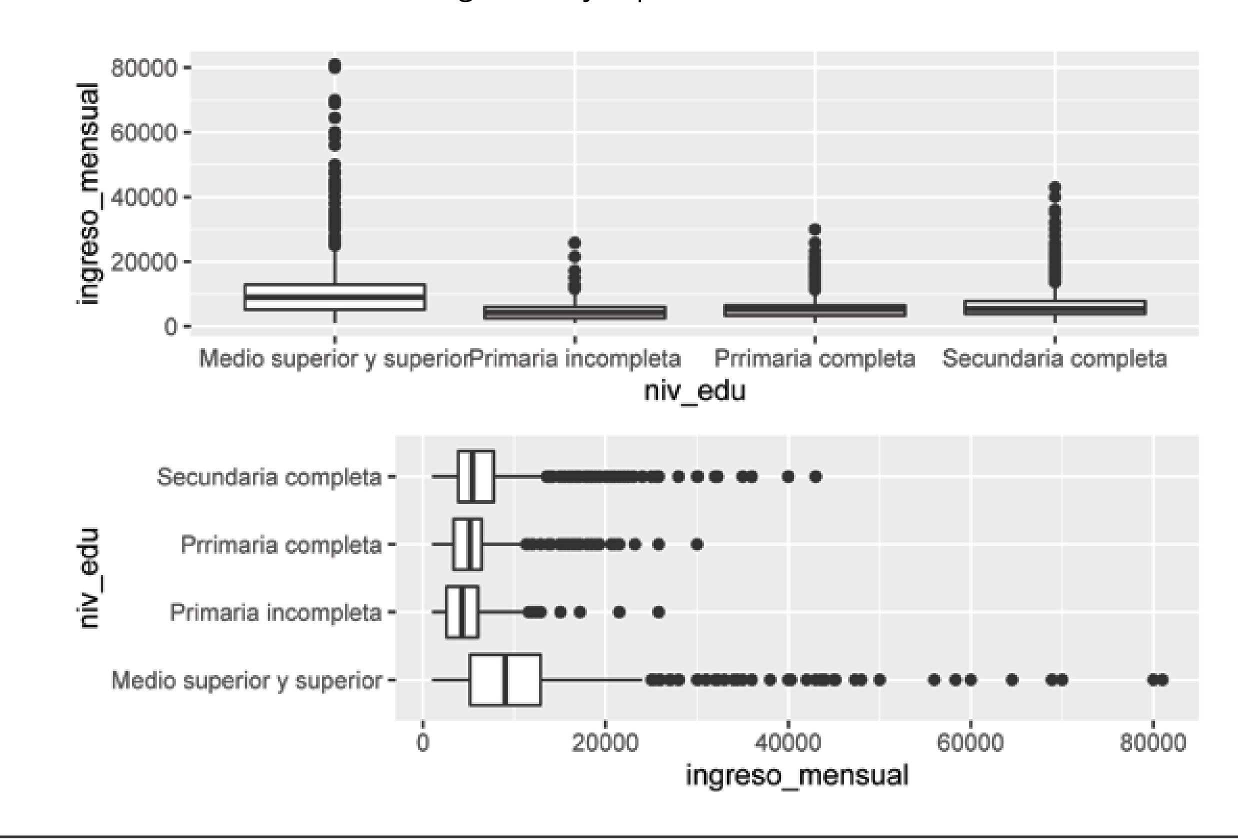
5.7 Sistema de coordenadas

De manera predeterminada **ggplot2** cuenta con un sistema de coordenadas cartesianas (x,y). Existen otros sistemas de coordenadas queayudan a construir gráficas más interesantes.

5.7.1 coord_flip()

coord_flip() nos permite hacer un cambio en las coordenadas x, y de la gráfica de manera que la información del eje x, se transforma en la información del eje y, y viceversa.

Veamos su funcionamiento con el siguiente ejemplo;



```
# superior
ggplot(data=enoe, mapping = aes(x=niv_edu, y=ingreso_mensual))+
     geom_boxplot()

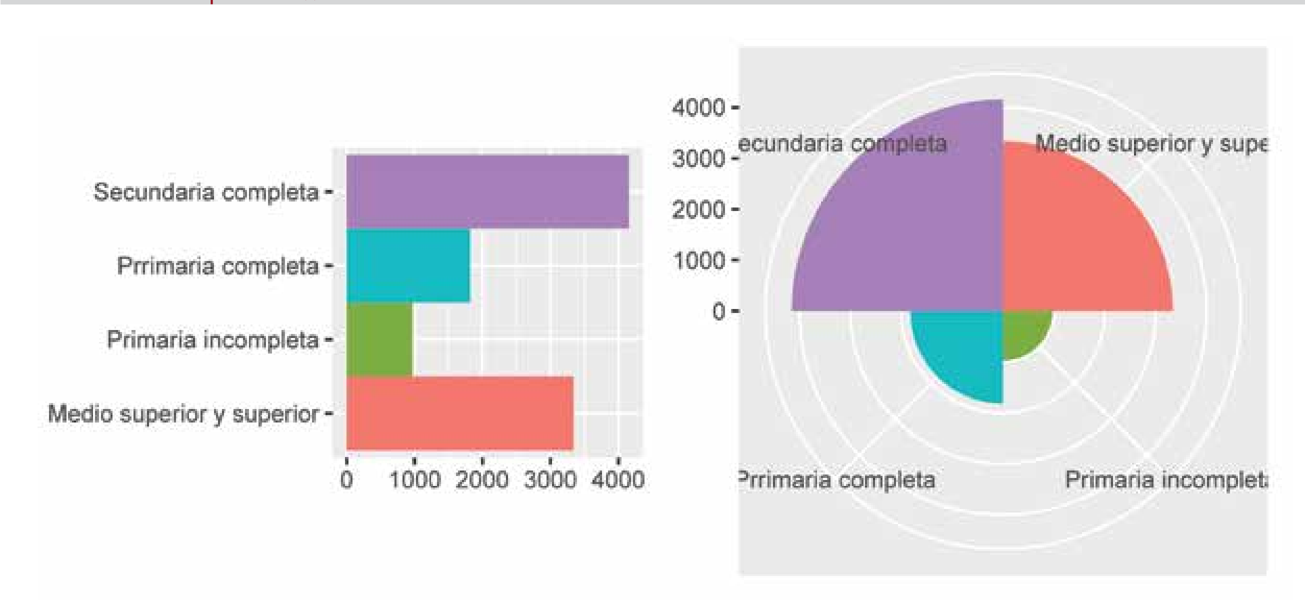
# inferior
ggplot(data=enoe, mapping = aes(x=niv_edu, y=ingreso_mensual))+
     geom_boxplot()+
     coord_flip()
```

Observa que en este caso hemos incluido la indicación sobre la variable que deseamos mapear dentro de las especificaciones de **ggplot()**, con el único fin de familiarizarnos con las distintas formas de especificar la instrucción.

5.7.2 coord_polar()

coord_polar() permite el uso de coordenadas polares. Consideremos el siguiente código que genera las gráficas;

```
barra←ggplot(data=enoe)+
     geom_bar(mapping = aes(x=niv_edu, fill=niv_edu),
          show.legend = FALSE, width = 1)+
     theme(aspect.ratio = 1)+
     labs(x=NULL, y=NULL)
barra+coord_flip()
barra+coord_polar()
```



Observa que en este caso hemos asignado al objeto **barra** una gráfica de barras. Es decir le hemos asignado un nombre a la gráfica. Además, hemos incluido dentro de las características **width = 1**, que se refiere al ancho de las barras, **theme(aspect.ratio = 1)** y **labs(x=NULL, y=NULL)**. Y una vez que se ha creado el objeto **barra**, se ha dado la indicación de construir dos gráficas. La primera combinando las especificaciones en el objeto **barra** mas el sistema de coordenadas **flip** y otra con el sistema de coordenadas **polar.**

Crear una gráfica y guardarla dentro de un objeto nos permite simplificar el código, ya que de otro modo, tendríamos que haber escrito;

```
# Gráfica superior
ggplot(data=enoe)+
     geom_bar(mapping = aes(x=niv_edu, fill=niv_edu),
          show.legend = FALSE, width = 1)+
     theme(aspect.ratio = 1)+
     labs(x=NULL, y=NULL)+
     coord_flip()
# Gráfica inferior
ggplot(data=enoe)+
     geom_bar(mapping = aes(x=niv_edu, fill=niv_edu),
          show.legend = FALSE, width = 1)+
     theme(aspect.ratio = 1)+
     labs(x=NULL, y=NULL)+
     coord_polar()
```

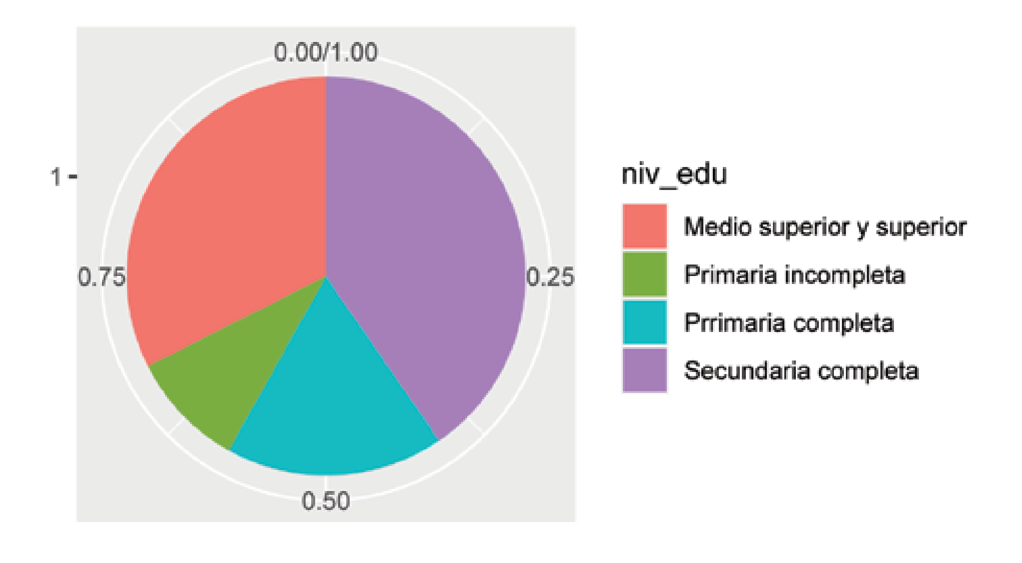
Puedes ejecutar este código para comprobar que obtenemos el mismo resultado.

Supongamos ahora que queremos ahora ver un gráfico circular donde sea posible observar los porcentajes de observaciones de cada uno de los diferentes niveles educativos.

En este caso, las gráficas circulares con **ggplot2**, parten de la construcción de una gráficas de barras apiladas, únicamente con un cambio de coordenadas.

```
ggplot(data=enoe, mapping = aes(x=factor(1), fill=niv_edu))+
     geom_bar(position = "fill")+
     coord_polar(theta = "y") +
     labs(x="", y="")
```

Nota que hemos usado la indicación **theta = "y"** indicando que sobre la variable y, el equivalente al conteo de los datos, es sobre la cual deseamos efectuar los ángulos (o bien, las diferentes divisiones) de la gráfica.



6 La gramática de ggplot2

Iniciamos este capitulo con una construcción básica para graficar con **ggplot2**, que incluía únicamente tres elementos. Hasta ahora hemos añadido cuatro elementos mas que permiten mejor visualización de los datos **ggplot2**. Esto totaliza siete parámetros, cuya estructura general responde a lo siguiente

- ggplot(data = **DATOS**) + **GEOM_FUNCIÓN**(mapping = aes(**MAPEOS**), stat = **ESTADÍSTICAS**, position = **POSICIÓN**) + **FUNCIÓN_COORDENADAS** + **FUNCIÓN_FACETAS**

Estos parámetros componen la **gramática** de un gráfica con **ggplot2**. En consecuencia, una gráfica se puede describir como una combinación de un conjunto de datos, una geometría, un conjunto de mapeos, una transformación estadística, una posición, un sistema de coordenadas y un esquema de facetas. Esta plantilla puede ser utilizada para crear una amplia variedad de gráficas.

Siempre que desees elaborar una gráfica con **ggplot** debes tener en cuenta la estructura de la información con la que cuentas. Por ejemplo, si usamos la base de datos **enoe** y queremos representar el promedio de ingreso mensual de las mujeres y de los hombres en la muestra, es necesario primero obtener las estadísticas, si bien el uso de **gem_boxplot()**, nos da una idea, no nos permite ver con claridad, las diferencias. En los siguientes capítulos aprenderemos mas sobre el manejo de datos, lo que permitirá ampliar los datos que queramos representar. Además, en el capítulo 17 aprenderemos mas sobre como cambiar el estilo y el diseño de una gráfica con el fin de que podamos comunicar de una mejor manera el comportamiento de los datos.

7 Actividades

1. Elabora una gráfica de dispersión que permita observar la relación entre el salario mensual y la edad. Donde todos los puntos sean rojos.

2. Elabora una gráfica que permita observar la relación entre el ingreso entre el salario mensual y la edad. Además que los puntos permitan identificar el nivel de escolaridad.

3. Intenta efectuar una gráfica de puntos, donde en el eje x se observen los años de educación, en el eje y, el ingreso mensual y ademas usando la estética de alpha sobre la variable nivel de educación. ¿Cuál es el warning que emite **R**? ¿Porqué crees que sucede esto?

4. Elabora una gráfica que permita ver la relación del ingreso mensual con los años de educación diferenciada por sexo, de manera que cambie no sólo el color del punto, sino ademas la forma. ¿Qué información relevante aporta esa gráfica?

5. Construye las siguientes gráficas, analiza detenidamente las diferencias entre ellas y responde las preguntas.

- ggplot(data = enoe) + geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual), shape=3, size=3, color="red")
- ggplot(data = enoe) + geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual), shape=5, size=1, color="blue")

- ¿Qué impacto tiene sobre la gráfica la opción **shape**, **size** y **color** cuando éstas se incluyen fuera de la función **aes**?

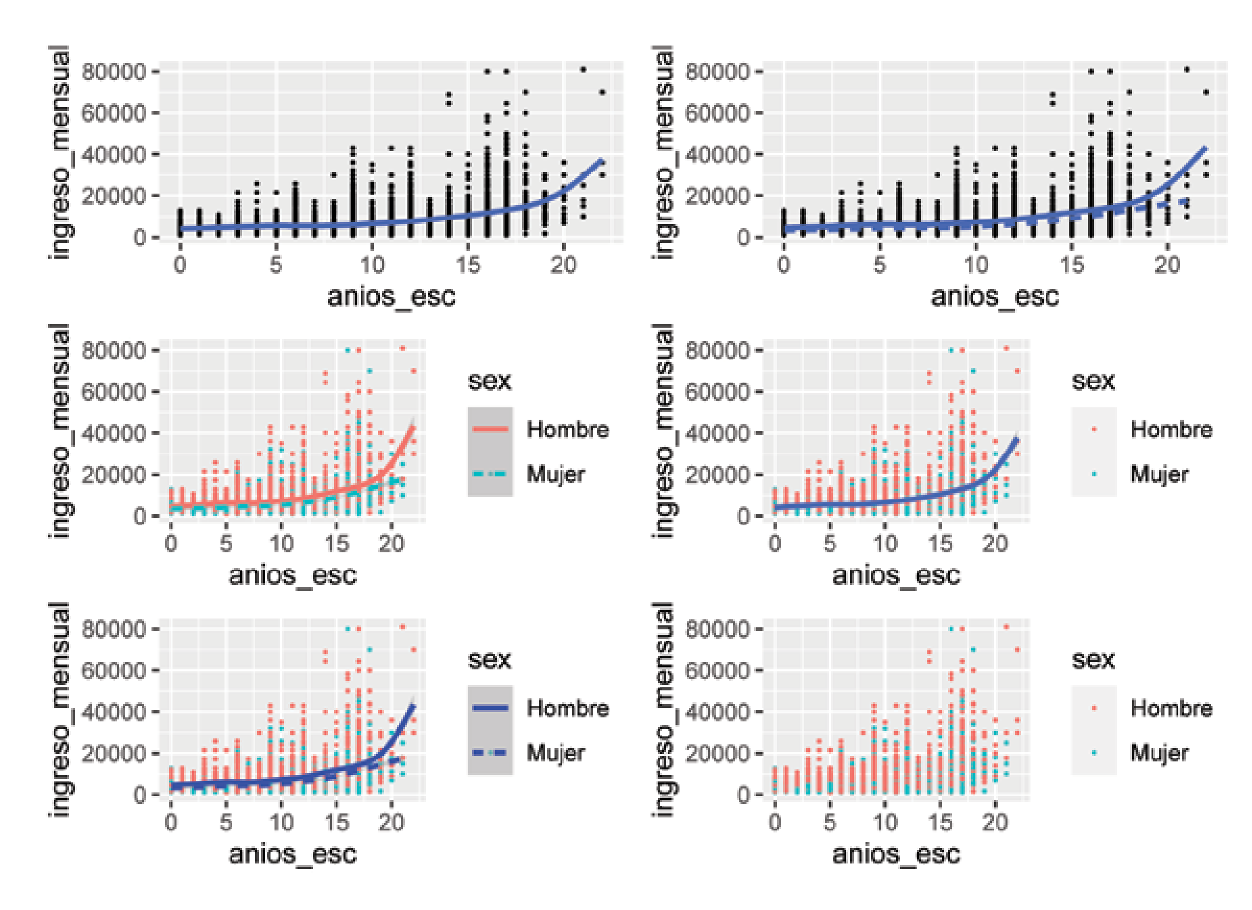
6. Elabora una gráfica que permita observar la relación la edad y el ingreso mensual, diferenciando por estados usando primero **color** y después **shape**. ¿Observas alguna inconsistencia?

7. ¿Qué gráficas generan los siguientes códigos?

- ggplot(data=enoe)+ geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))+ facet_grid(sex~.)
- ggplot(data=enoe)+ geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))+ facet_grid(. ~ tipo_empleo)
- ¿Que función tiene "." dentro de código?

8. Replica la gráfica para separa Jalisco, pero elimina la opción **se**. ¿Que función tiene **se**?

9. ¿Cuál sería el código necesario para construir cada una de las siguientes gráficas?



10. ¿Cuál es la diferencia entre geom_col y geom_bar?

11. Una de las geometrias en ggplot2 es geom_jitter, la cual crea el mismo efecto que position="jitter" ¿Cómo sería el código para construir la misma gráfica usando geom_jitter en lugar de position="jitter"?



Capítulo 4

Transformación y exploración de datos

Capítulo 4 | Transformación y exploración de datos

1 Introducción

La visualización nos ayuda a entender de manera más intuitiva la información. En muchas ocasiones los datos que utilizamos no tienen la forma adecuada para generar las gráficas deseadas. En ocasiones, necesitamos crear nuevas variables, generar agregados, renombrar variables, reordenar o reclasificar observaciones.

En este apartado aprenderemos a realizar estas operaciones con la finalidad de que podamos transformar una base de datos según nuestras necesidades de información, para ello haremos uso de la paquetería **dplyr**.

Algunas de las herramientas más útiles que usaremos las encontraremos en esta paquetería, con las cuales podemos filtrar, tanto observaciones como variables, generar nuevas variables, obtener un resumen estadístico de las variables de interés y combinar estas funciones para obtener de manera rápida y eficaz información de nuestros datos. Es importante aprender el uso de estas funciones ya que en la práctica muchas de nuestras fuentes de información no siempre tienen las variables que nos interesan por lo que debemos manipular la base de datos, eliminar variables innecesarias, acomodar u ordenar observaciones, etc. En la primera parte de este capítulo nos enfocaremos en este fin.

En la segunda parte, aprenderemos a realizar un análisis exploratorio de datos utilizando **R**. Este proceso es importante ya que nos permite identificar algunas relaciones entre las variables. El análisis exploratorio permite que nos hagamos preguntas y tratar de encontrar respuestas a estas preguntas.

Antes de comenzar a manipular datos, es necesario comenzar nuestro script utilizando los consejos que se dieron en capítulos anteriores:

- Comenzar un script nuevo
- Establecer un directorio de trabajo
- Limpiar el entorno de trabajo

1.2 Prerrequisitos

En este apartado haremos uso de las paquetería **dplyr, tidyverse, ggplot2 y readr**. Anteriormente aprendimos a importar una base de datos de Excel, en esta ocasión vamos a importar una base de datos de un archivo **"csv"**.

En este apartado trabajáremos con la información de la "Encuesta Nacional de Ingresos y Gastos de los Hogares" (ENIGH), la cual realiza el Instituto Nacional de Estadística y Geografía (INEGI). En dicha encuesta podemos encontrar los patrones de ingreso y gasto de los hogares mexicanos, así como información adicional sobre las características sociodemográficas de los integrantes del hogar.

Al igual que antes, el primer paso consiste en instalar las paqueterías necesarias en caso de no contar con ellas. Una vez instaladas debemos cargarlas para que estén disponibles y trabajar con ellas. Al igual que antes necesitamos establecer un directorio de trabajo. Recuerda utilizar la carpeta donde se encuentra el archivo **hogares_enigh.csv**.

Añadiremos a nuestro entorno la siguiente instrucción options(scipen=999), la cual desactivara la notación científica, pues a pesar de que no se estén manejando números grandes, **R** de manera predeterminada trata de simplificar la visualización de números, sin embargo, a veces esto puede ser un inconveniente, por el momento desactivaremos esta notación.

```
setwd("~/Dropbox/Curso de R")
library (tidyverse)
options(scipen=999)
```

Ahora podemos importar la base de datos a un objeto en **R** al cual llamaremos **enigh**:

```
enigh <- read_csv("hogares_enigh.csv")
##
## -- Column specification ------------------------------------------------
## cols(
## .default = col_double(),
## folioviv = col_character(),
## ubica_geo = col_character(),
## est_dis = col_character(),
## upm = col_character(),
## educa_jefe = col_character()
## )
## i Use `spec()` for the full column specifications.
```

```
enigh
## # A tibble: 74,647 x 26
## folioviv foliohog ubica_geo tam_loc est_socio est_dis upm factor clase_hog
## <chr> <dbl> <chr> <dbl> <dbl> <chr> <chr> <dbl> <dbl>
## 1 0100013~ 1 01001 1 3 002 0000~ 175 2
## 2 0100013~ 1 01001 1 3 002 0000~ 175 2
## 3 0100013~ 1 01001 1 3 002 0000~ 175 2
## 4 0100013~ 1 01001 1 3 002 0000~ 175 2
## 5 0100013~ 1 01001 1 3 002 0000~ 175 2
## 6 0100026~ 1 01001 1 3 002 0000~ 189 2
## 7 0100026~ 1 01001 1 3 002 0000~ 189 1
## 8 0100026~ 1 01001 1 3 002 0000~ 189 2
## 9 0100026~ 1 01001 1 3 002 0000~ 189 2
## 10 0100027~ 1 01001 1 3 002 0000~ 186 3
## # ... with 74,637 more rows, and 17 more variables: sexo_jefe <dbl>,
## # edad_jefe <dbl>, educa_jefe <chr>, tot_integ <dbl>, percep_ing <dbl>,
## # ing_cor <dbl>, ingtrab <dbl>, gasto_mon <dbl>, alimentos <dbl>,
## # vesti_calz <dbl>, vivienda <dbl>, limpieza <dbl>, salud <dbl>,
## # transporte <dbl>, educa_espa <dbl>, personales <dbl>, transf_gas <dbl>
```

En la descripción que arroja el **data frame** observamos en la primera fila los nombres de las columnas, y debajo una descripción sobre el tipo de variable de que se trata:

- int: números enteros
- dbl: números reales
- chr: caracter o texto
- dttm: variable de tiempo
- lgl: valores lógicos, verdadero o falso
- fctr: factores
- date: fechas

1.3 Funciones básicas de dplyr

La paquetería (o librería) **dplyr** tiene cinco funciones principales:

- filter(): filtrar observaciones (filas)
- arrange(): ordenar observaciones
- select(); filtrar variables (columnas)
- mutate(): crear nuevas variables usando un data frame
- summarize(): comprimir muchos valores en un solo estadístico

Estas funciones pueden combinarse con la opción **group_by()**, la cual permite separar el análisis por grupos según nuestras necesidades. Por ejemplo, podríamos manipular la información por entidades o estrato urbano y rural, niveles de ingreso, etc.

Estas cinco funciones tienen una sintaxis similar:

- 1. El primer argumento el nombre del objeto tipo data frame
- 2. El segundo argumento describe lo que deseemos hacer con el data frame
- 3. El resultado, consistente con el nombre donde pondremos el nuevo data frame, resultante de la manipulación de datos

1.3.1 Filtrar datos

En relación a la función **filter()** permite hacer un subconjunto de observaciones basado en los valores de interés. La función permite utilizar una o más condiciones. De esta forma el resultado que obtendremos serán aquellas observaciones donde se cumplan las restricciones que coloquemos.

Por ejemplo, podemos filtrar los hogares por el tipo de la clase de hogar **(clase_hog)** según su parentesco.

Filtremos los hogares de tipo nuclear, cuya clasificación es 2, esto se obtiene haciendo **clase_hog==2.**

```
hogares_nucleares <- filter(enigh, clase_hog==2)
```

Observa que para el correcto funcionamiento de esta instrucción utilizamos un doble signo de igualdad "==", esto se debe al tipo de operador que requiere la función.

Un operador es un símbolo que nos ayuda a comunicarle al software determinada acciones. Existen varios tipos de operadores, hasta ahora hemos usado el operador asignación < -, sin embargo, el símbolo = es también un operador de asignación. En cambio, algunas funciones requieren de un operador de comparación.

Tipos de operadores

- Aritméticos
  - Suma +
  - Resta -
  - Multiplicación *
  - División /
- Comparativos
  - Menor que <
  - Mayor que >
  - Menor o igual que <=
  - Mayor o igual que >=
  - Igual ==
  - Diferente que !=
- Lógicos
  - y lógico &
  - o lógico |
- Asignación
  - < -
  - =

En nuestro ejemplo, requerimos de un operador de comparación, en ese caso el de igualdad.

Ahora utilicemos varias combinaciones para filtrar los datos, por ejemplo, hogares nucleares y con jefatura femenina. Para ello identificamos primero que la jefatura femenina es aquella que corresponde a las observaciones donde la variable sexo_jefe toma el valor de 2.

```
hogares_nucleares_jfem <- filter(enigh, clase_hog==2, sexo_jefe==2)
```

Observa que hemos indicado **dos condiciones, cada una en diferente variable** que deben cumplirse para cada una de las observaciones que vamos a filtrar.

Otro ejemplo donde utilicemos más de un operador comparativo es cuando deseamos filtrar **más de un valor** en una **misma variable**. Filtremos los hogares de tipo unipersonales y nucleares con jefatura femenina. En este caso los hogares unipersonales se identifican cuando la variable **clase_hog** toma el valor de 1.

En consecuencia:

```
hogares_jfem <- filter(enigh, (clase_hog==1 | clase_hog==2) , sexo_jefe==2)
```

Observa que ahora hemos establecido dos condiciones, sobre la misa variable, por lo que hemos utilizado el operador |, el cual corresponde con la **o lógica**. También hemos agregado los paréntesis, para asegurar el cumplimiento de esa condición y separarla de la condición siguiente.

En cada una de las instrucciones que hemos efectuado, hemos asignado un nombre al objeto resultante de la operación. Te invitamos a que los explores y veas su contenido. Para abrir estos objetos y ver su contenido, pues escribir **View(nombreobjeto)**. Esto abrirá una pequeña venta junto a la pestaña donde esta tu script. Ahí te puedes dar cuenta y analizar el contenido de la base, de esta forma tendrás mayor claridad sobre el funcionamiento de estos comandos.

1.3.2 Ordernar observaciones usando arrange()

La función **arrange()** cambia el orden de las observaciones, podemos asignar el orden según el valor de una o más columnas. El orden jerárquico corresponderá al mismo que describamos en la instrucción, es decir, si tenemos una base de datos con nombres de personas: "Apellido_paterno", "Apellido_materno" y "Nombre", y utilizamos la función **arrange()**, para ordenar los nombres por apellido paterno, seguido del materno y el nombre, debemos usar la siguiente instrucción:

```
arrange(data_frame, Apellido_parterno, Apellido_materno, Nombre)
```

Probemos esta indicación ordenando la base enigh según la edad del jefe del hogar:

```
arrange(enigh, edad_jefe)
```

Ahora si deseamos ordenar primero por el sexo del jefe del hogar y después la edad:

```
arrange(enigh, sexo_jefe, edad_jefe)
```

De manera predeterminada, la función arrange() ordena los valores de **menor a mayor**, sin embargo, podemos especificar si deseamos que el orden descendente, para ello debemos poner especificar **desc()** y dentro del paréntesis la variable por la cual deseamos reordenar las observaciones:

```
arrange(enigh, sexo_jefe, desc(edad_jefe))
```

En caso de existir valores perdidos en la variable, estos vienen identificados con un “NA” que significa “No disponible”, **R** los colocará siempre al final independientemente si se ordena de manera ascendente o descendente. Más tarde discutiremos sobre los valores perdidos.

1.3.3 Filtrar variables con select()

En muchas ocasiones encontraremos que nuestra base de datos contiene muchas variables, más de las requerimos para nuestro análisis. Por ejemplo, la base con la que estamos trabajando contiene 26 variables. Supongamos que deseamos explorar de manera rápida algunas de ellas, para ello podemos utilizar la función select() que permite separar o filtrar variables por su nombre.

Por ejemplo, en la encuesta enigh, los ingresos y gastos están desagregados en diferentes niveles, pero no nos interesa, por ahora, explorar a tal nivel de desagregación, por lo que solo queremos explorar los datos generales del hogar y el total de ingreso y gasto:

Para explorar los nombres de las variables utilizamos la siguiente instrucción:

```
colnames(enigh)
## [1] "folioviv"     "foliohog"   "ubica_geo"    "tam_loc"    "est_socio"
## [6] "est_dis"      "upm"        "factor"       "clase_hog"  "sexo_jefe"
## [11] "edad_jefe"   "educa_jefe" "tot_integ"    "percep_ing" "ing_cor"
## [16] "ingtrab"     "gasto_mon"  "alimentos"    "vesti_calz" "vivienda"
## [21] "limpieza"    "salud"      "transporte"   "educa_espa" "personales"
## [26] "transf_gas"
```

Observa que las primeras variables corresponden a información general del hogar, la variable 15 “ing_cor” es la variable de ingreso del hogar y la variable 17 “gasto_mon” la variable referente al gasto. Para este ejemplo conservemos únicamente estas variables, acompañadas de algunos datos generales del hogar, para ello debemos nombrar cada una de las variables que deseemos conservar.

```
enigh_corto <- select(enigh, folioviv, foliohog,
      ing_cor,gasto_mon, tot_integ,
      ubica_geo, sexo_jefe, clase_hog,edad_jefe)
```

En el caso que deseemos conservar por ejemplo desde la variable “folioviv” hasta “sexo_jefe” y además el “gasto_mon”, debemos hacer lo siguiente

```
enigh_corto2 <- select(enigh, folioviv:sexo_jefe, gasto_mon)
```


En este caso para evitar escribir cada una de ellas podemos utilizar el símbolo “:” para indicar inclusión. “folioviv:sexo_jefe” indica incluir desde folioviv hasta sexo_jefe

Ahora el data frame que llamamos “enigh_corto” contiene solo 9 variables, las cuales podemos explorar de manera más rápida:

```
View(enigh_corto)
```

También podemos usar el símbolo “:” para indicar exclusión, por ejemplo, excluir las variables desde “folioviv” hasta “percep_ing”, usamos la siguiente instrucción:

```
select(enigh, -(folioviv:percep_ing))
## # A tibble: 74,647 x 12
## ing_cor ingtrab gasto_mon alimentos vesti_calz vivienda limpieza salud
##    <dbl>    <dbl>    <dbl>    <dbl>  <dbl> <dbl>    <dbl>    <dbl>
## 1  76404.    53115.  18551.    5618.  0    3912     522      0.
## 2  42988.    15235.  55471.   20930.  401. 2495     412.     1.35e3
## 3 580698.   141885. 103107.   37594. 2015. 4475     3318.    2.89e4
## 4  46253.     0      19340.    2893.  97.8 1458     5514     3.23e2
## 5  53837.    43229.  13605.    7367.  0     300     3300     5.67e1
## 6  237743.   129836. 33628.    0      0     2801    5682     0.
## 7  32607.    23607.  33311.   11456.  0     4405    2497.    4.70e3
## 8  169918.   154918. 87119.   30986. 1565. 1050     10376.   0.
## 9  17311.    17311.  14347.    5773.  533. 4740      771     4.89e1
## 10 120488.   44761.  52373.    3986. 1174. 16199.   784.     8.22e3
## # ... with 74,637 more rows, and 4 more variables: transporte <dbl>,
## # educa_espa <dbl>, personales <dbl>, transf_gas <dbl>
```

En este caso hemos utilizado el signo - para indicar que no deseamos esas variables en la base filtrada.

Existen algunas otras funcionen que podemos utilizar junto con select():

- starts_with(“abc”) selecciona las variables que comienzan con “abc”
- ends_with(“xyz”) selecciona las variables que terminan con “xyz”
- contains(“ijk”) selecciona las variable que en su nombre contienen “ijk”
- matches(“(.)\1”) selecciona las variables que corresponden a una expresión regular
- num_range(“x”, 1:3) selecciona las variables llamadas x1, x2 y x3

Por ejemplo;

```
select(enigh, starts_with("fol"))
## # A tibble: 74,647 x 2
## folioviv foliohog
## <chr>             <dbl>
## 1 0100013601        1
## 2 0100013602        1
## 3 0100013603        1
## 4 0100013604        1
## 5 0100013606        1
## 6 0100026701        1
## 7 0100026703        1
## 8 0100026704        1
## 9 0100026706        1
## 10 0100027201       1
## # ... with 74,637 more rows
```

```
select(enigh, ends_with("s"))
## # A tibble: 74,647 x 4
## est_dis alimentos personales transf_gas
##         <chr> <dbl>      <dbl>      <dbl>
##    1    002   5618.        99       0
##    2    002   20930.     4663.      24.6
##    3    002   37594.     8520       6000
##    4    002   2893.      1065.      0
##    5    002   7367.      1686.      295.
##    6    002      0       5109       0
##    7    002   11456.     3336.      492.
##    8    002   30986.     2136       23607.
##    9    002   5773.      1125       0
##    10   002   3986.      3460.      2919.
## # ... with 74,637 more rows
```

Observa que en todos los casos select() permite filtrar variables.

1.3.4 Generar nuevas variables con mutate()

En muchas ocasiones tendremos que generar nuevas variables, mutate() es la función que nos ayudará para esta tarea. Esta función añadirá nuevas columnas a un conjunto de datos existente.

Antes de continuar limpiemos nuestro ambiente de trabajo, conservemos solo los data frame enigh y enigh_corto.

```
remove (hogares_nucleares, hogares_nucleares_jfem, hogares_jfem, enigh_cor-
to2)
```

Ahora generaremos tres variables sobre el data frame enigh_corto: el logaritmo natural del ingreso corriente, el logaritmo natural del gasto monetario y el gasto monetario como porcentaje del ingreso:

```
enigh_corto <- mutate(enigh_corto, lingreso=log(ing_cor),
                 lgasto=log(gasto_mon),
                 gasto_porcentaje=(gasto_mon/ing_cor)*100 )
```

Observa que en cada caso hemos definido una operación sobre una variable y hemos asignado un nuevo nombre para esa variable. Esto se observa en la estructura de las indicaciones, cómo por ejemplo; `lingreso=log(ing_cor)`.

A través de la función **mutate()** podemos generar variables con diferentes operaciones aritméticas, transformaciones logarítmicas, jerarquías, etc.

Veamos otro ejemplo. Ahora incluyamos también el ingreso per cápita del hogar, para ello debemos dividir el ingreso del hogar entre el número de habitantes del hogar, esto para tener un mejor indicador del nivel de ingreso del hogar:

```
enigh_corto <- mutate(enigh_corto, ingreso_capita=ing_cor/tot_integ )
```

La encuesta enigh, es representativa a por entidad federativa, sin embargo, para conocer que hogares representan a cada entidad, debemos sustraer los primeros dos dígitos de la variable "ubica_geo", estos primeros dos dígitos son la clave de la entidad federativa donde se encuentra el hogar. Por ejemplo, si la variable ubica_geo es igual a 01001, significa que se trata de un hogar que se encuentra en el estado 01, el cual, de acuerdo con la clasificación de hogares en México, corresponde a Aguascalientes.

Para realizar esta tarea debemos usar la función mutate() y la función substr().

```
enigh_corto <- mutate(enigh_corto, cve_ent=substr(ubica_geo,1,2) )
```

La función substr() sustrae desde el primer carácter (por eso indicamos 1) hasta el segundo (por eso indicamos 2) de la variable "ubica_geo".

1.3.5 Función summarise()

La función summarize() es útil para generar un resumen de estadísticos (media, mediana, varianza, etc.) de una base de datos.

```
summarise(enigh_corto, mean(ing_cor), mean(gasto_mon))
## # A tibble: 1 x 2
## `mean(ing_cor)` `mean(gasto_mon)`
##           <dbl>             <dbl>
## 1        46044.            28990.
```

```
summarise(enigh_corto, median(ing_cor), median(gasto_mon))
## # A tibble: 1 x 2
## `median(ing_cor)` `median(gasto_mon)`
##           <dbl>               <dbl>
## 1        33573.              22116.
```

```
summarise(enigh_corto, var(ing_cor), var(gasto_mon))
## # A tibble: 1 x 2
## `var(ing_cor)` `var(gasto_mon)`
##           <dbl>            <dbl>
## 1  3749609091.       808587884.
```

En el primer ejemplo hemos calculado la media (mean), de cada una de las variables indicadas. En el segundo la mediana (median) y en el tercer caso la varianza (var)

Sin embargo, esta función tiene mayor utilidad cuando la combinamos con la opción group_by(), de esta manera obtenemos un grupo de estadísticos para diferentes grupos de observación. Por ejemplo, veamos el promedio de ingreso y gasto según el sexo del jefe del hogar.

Primero generamos el grupo de sexo de jefe del hogar y después calculamos los promedios para cada grupo:


```
sexo <- group_by(enigh_corto, sexo_jefe)
summarise ( sexo, mean(ing_cor), mean(gasto_mon))
## `summarise()` ungrouping output (override with `.groups` argument)
## # A tibble: 2 x 3
## sexo_jefe `mean(ing_cor)` `mean(gasto_mon)`
##       <dbl> <dbl>       <dbl>
## 1         1     47852.      30082.
## 2         2     41261.      26101.
```

Observa que primero se ha efectuado la agrupación por sexo, después se ha creado la media cada grupo.

Otra forma de hacer el mismo cálculo es integrando en una sola instrucción la generación de los grupos:

```
summarise ( group_by(enigh_corto, sexo_jefe),
mean(ing_cor), mean(gasto_mon))
## `summarise()` ungrouping output (override with `.groups` argument)
## # A tibble: 2 x 3
## sexo_jefe `mean(ing_cor)` `mean(gasto_mon)`
##           <dbl>       <dbl>      <dbl>
## 1         1     47852.      30082.
## 2         2     41261.      26101.
```

Si estas prestando atención a tu código observarás que en ambos casos **R** nos muestra la leyenda *summarise() ungrouping output (override with.groupsargument)*. Esto sucede porque la funciónsummarise()espera que le indiquemos que hacer con la agrupación después de obtener los estadísticos solicitados. Si no le damos alguna indicación de forma automática los elimina. Si no queremos ver este mensaje, podemos agregar la indicación.groups = 'drop", para indicarle que puede eliminar la agrupación efectuada.

Este método, puede generar confusión a la ahora de revisar el código, si los nombres de las variables son muy largos. Además, si los grupos se usarán con relativa frecuencia, conviene entonces utilizar el primer enfoque.

En este punto resulta importante señalar lo siguiente. Con la indicación **sexo <- group_by(enigh_corto, sexo_jefe)** hemos creado un grupo, del cual debes observar lo siguiente:

```
class(sexo)
## [1]  "grouped_df"  "tbl_df"   "tbl"      "data.frame"
```

```
dim(sexo)
```

```
## [1] 74647     14
```

Este grupo tiene la misma dimensión que enigh, incluso si abres ambos objetos para analizarlos detalladamente, te darás cuentas que parecen iguales, es decir, tienen exactamente la misma información. La diferencia fundamental es que el primero es una copia de la enigh que ha sido agrupada por sexo, aún y cuando esa agrupación no sea visible para nosotros dentro del objeto.

Calculemos ahora las diferencias por clase de hogar y sexo del jefe del hogar:

```
clases_hogar ← group_by(enigh_corto, sexo_jefe, clase_hog)
summarise ( clases_hogar, mean(ing_cor), mean(gasto_mon), .groups = 'drop')
```

```
## # A tibble: 10 x 4
## sexo_jefe clase_hog `mean(ing_cor)` `mean(gasto_mon)`
## <dbl>    <dbl> <dbl>       <dbl>
## 1  1      1     30809.      20246.
## 2  1      2     47397.      30238.
## 3  1      3     55619.      33228.
## 4  1      4     68479.      39422.
## 5  1      5     78385.      49456.
## 6  2      1     26019.      15211.
## 7  2      2     41463.      27451.
## 8  2      3     48832.      29765.
## 9  2      4     67079.      41260.
##10  2      5     64633.      47255.
```

Ahora podemos ver los promedios de ingreso y gasto por entidad federativa.

```
entidades ← group_by(enigh_corto, cve_ent)
summarise ( entidades, mean(ing_cor), mean(gasto_mon),.groups = 'drop')
```

```
## # A tibble: 32 x 3
## cve_ent `mean(ing_cor)` `mean(gasto_mon)`
## <chr>    <dbl>      <dbl>
## 1  01     57622.     34993.
## 2  02     51244.     33422.
```



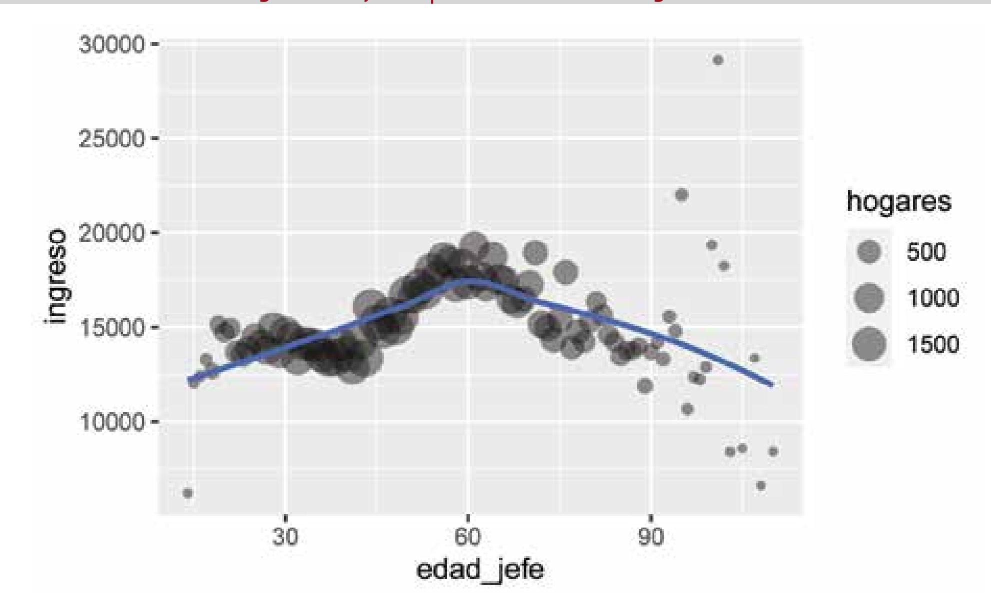
```
## 3  03    64987.      35293.
## 4  04    48877.      31268.
## 5  05    51082.      31018.
## 6  06    46539.      29366.
## 7  07    26107.      18782.
## 8  08    48850.      26031.
## 9  09    67466.      40146.
## 10 10    40597.      26964.
## # ... with 22 more rows
```

¿Existen una relación entre el ingreso y la edad del jefe del hogar? exploremos un poco más los datos. Existen hogares que declaran cero ingresos o se rehusaron a dar información, es mejor filtrar estos hogares. Lo mismo debemos hacer con el gasto.

```
enigh_corto_no_ceros ← filter(enigh_corto, ing_cor>0, gasto_mon>0 )
edades ← group_by(enigh_corto_no_ceros, edad_jefe)
ingreso_edad ← summarise ( edades, ingreso=mean(ingreso_capita), hoga-
res=n(), .groups = 'drop' ) edades
```

```
## # A tibble: 74,574 x 14
## # Groups: edad_jefe [94]
## folioviv foliohog ing_cor gasto_mon tot_integ ubica_geo sexo_jefe clase_hog
## <chr>         <dbl> <dbl>      <dbl>      <dbl> <chr>      <dbl> <dbl>
## 1 0100013~    1     76404.     18551.     3     01001      1     2
## 2 0100013~    1     42988.     55471.     5     01001      1     2
## 3 0100013~    1     580698.    103107.    2     01001      1     2
## 4 0100013~    1     46253.     19340.     2     01001      2     2
## 5 0100013~    1     53837.     13605.     4     01001      2     2
## 6 0100026~    1     237743.    33628.     4     01001      2     2
## 7 0100026~    1     32607.     33311.     1     01001      2     1
## 8 0100026~    1     169918.    87119.     2     01001      1     2
## 9 0100026~    1     17311.     14347.     3      01001     1     2
## 10 0100027~   1     120488.    52373.     4     01001      1     3
## # ... with 74,564 more rows, and 6 more variables: edad_jefe <dbl>,
## #  lingreso <dbl>, lgasto <dbl>, gasto_porcentaje <dbl>, ingreso_capita
<dbl>,
## # cve_ent <chr>
```

```
ggplot(data=ingreso_edad, mapping = aes(x=edad_jefe, y=ingreso)) +
geom_point(aes(size = hogares), alpha = 1/3) + geom_smooth(se=FALSE)
```



La primera indicación es para filtrar los datos, usando únicamente la información de los hogares que reportan ingresos mayores a cero. En la segunda línea estamos agrupando la información por edades, mientras que en la tercera estamos solicitando se calcule la media del ingreso per cápita. En este caso hemos acompañado la función **summarise** con la indicación **hogares=n()**, lo cual hará que se genere dentro de la base **ingreso_edad** una variable llamada **hogares** donde se contabilizará el total de hogares por grupo de edad.

Observamos que existe una relación en forma de U invertida con edad. Para esta muestra, los hogares alcanzan un máximo de ingresos cuando el jefe del hogar tiene al rededor de 60 años.

1.3.5.1 Operador pipe

Repasemos un poco las instrucciones para generar la gráfica que relaciona la edad del jefe del hogar con el ingreso.

1. Filtrar valores no creíbles como los ceros en las variables de ingreso y gasto
2. Genera los grupos de edad
3. Genera un nuevo data frame con el promedio de ingresos per cápita por edad y el número de hogares
4. Generar la gráfica

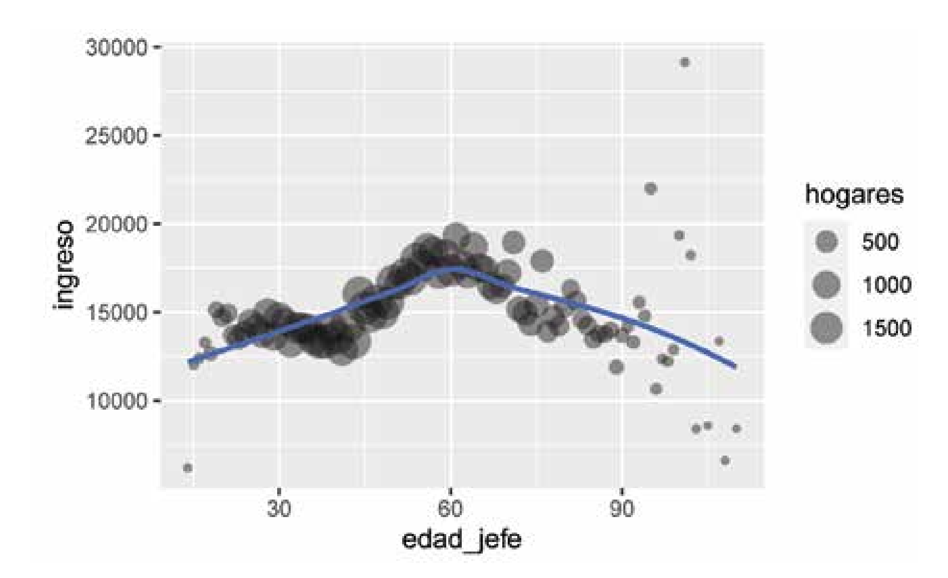
Si bien no es número grande de operaciones, puede generar confusión escribir líneas de código separadas para una misma tarea, existe un operador que nos puede ayudar a declarar una secuencia de tareas u operaciones.

Realicemos el mismo ejercicio, pero ahora con el uso del operador pipe %>%, el cual nos ayuda a concatenar o juntar una serie de operaciones o manipulaciones a un objeto en una misma instrucción:

```
ingreso_edad2 <- filter(enigh_corto, ing_cor>0, gasto_mon>0 ) %>%
```



```
     group_by(edad_jefe) %>%
     sumarise (ingreso=mean(ingreso_capita), hogares=n(),.groups = 'drop' )
ggplot(data=ingreso_edad2, mapping = aes(x=edad_jefe, y=ingreso)) +
geom_point(aes(size = hogares), alpha = 1/3) + geom_smooth(se=FALSE)
```



Con el uso del operador pipe podemos realizar operaciones de forma anidada sin necesidad de generar data frames intermedias y hacer el código más amigable para la lectura.

Veamos otro ejemplo, supongamos que ahora deseamos analizar el ingreso per cápita de los hogares por entidad federativa y por clase hogar:

```
filter(enigh_corto, ing_cor>0, gasto_mon>0 ) %>%
     group_by(cve_ent, clase_hog) %>%
     sumarise (ingreso=mean(ingreso_capita), hogares=n(), .groups = 'drop' )
```

```
## # A tibble: 160 x 4
##    cve_ent clase_hog ingreso hogares
##   <chr> <dbl>   <dbl>      <int>
## 1  01     1     30791.     201
## 2  01     2     18046.     1549
## 3  01     3     12280.     527
## 4  01     4     14565.     22
## 5  01     5     23006.     5
## 6  02     1     33728.     515
## 7  02     2     16326.     2033
## 8  02     3     13853.     737
## 9  02     4     16442.     18
## 10 02     5     18940.     14
```

```
## # ... with 150 more rows
```

Además del promedio, podemos utilizar la función summarise(), por ejemplo: • median() la mediana • sd() la desviación estándar • min () mínimo • max() máximo • quantile(x, 0.25) cuartiles • first() primera observación • last() última observación • nth() la n-ésima observación • count contar el número de observaciones

En el siguiente ejemplo estamos calculando diferentes estadisticas del ingreso corriente para cada estado.

```
tabla_entidad <- filter(enigh_corto, ing_cor>0, gasto_mon>0 ) %>% group_by(c-
ve_ent) %>%
summarise (ingreso_promedio=mean(ing_cor), ingreso_sd=sd(ing_cor),
     ingreso_p50=median(ing_cor), ingreso_min=min(ing_cor), ingreso_max=-
max(ing_        cor), .groups = 'drop')
tabla_entidad
```

```
## # A tibble: 32 x 6
##    cve_ent ingreso_promedio ingreso_sd ingreso_p50 ingreso_min ingreso_max
##    <chr> <dbl>       <dbl>       <dbl>       <dbl>       <dbl>
## 1  01    57644.      69704.      43240.      1874.       2310762
## 2  02    51307.      44410.      39443.      1461.       773383.
## 3  03    65004.      126280.     47952.      1790.       4501830.
## 4  04    48877.      60068.      33586.      681.        1595729.
## 5  05    51096.      49691.      39249.      2400        1196848.
## 6  06    46539.      42480.      35694.      1613.       671410.
## 7  07    26107.      26081.      18409.      1703.       265672.
## 8  08    48948.      80682.      34457.      978.        3778279.
## 9  09    67466.      87853.      44970       2473.       1614130.
## 10 10    40628.      35508.      30473.      978.        432827.
## # ... with 22 more rows
```

2 Exploración de datos

Ya que hemos aprendido a manipular datos, nos enfocaremos ahora a su exploración, para ello a lo largo de esta sesión, presentaremos algunas definiciones propias de la estadística únicamente con la finalidad de que sea claro el proceso de exploración. Para mayores detalles en temas estadísticos te recomendamos consultar cual libro de estadística.

2.1 Palabras clave

El objetivo del análisis exploratorio de los datos es desarrollar un entendimiento de nuestros datos. La mejor manera de hacer esto es usar preguntas que sirvan de guía para nuestra investigación. Al realizar preguntas, estas centran nuestra atención a una parte específica de nuestro conjunto de datos y nos ayudan a decidir que gráficas, modelos o transformaciones debemos hacer.

Es difícil realizar preguntas reveladoras o que detonen un punto central de una investigación si precisamente no conocemos a detalle nuestros datos. Sin embargo, la idea de este proceso es hacer preguntas sencillas, las cuales poco a poco relevarán nuevos aspectos de nuestros datos, lo cual nos permite generar nuevas preguntas.

Si bien, no hay una regla o guía escrita para comenzar nuestro análisis, un punto de partida es a través de dos preguntas:

- ¿Qué tipo de variación ocurre dentro de las variables?
- ¿Qué tipo de co-variación ocurre entre las variables?

Para tener un mejor entendimiento es necesario definir algunos conceptos:

- Variable: Se refiere a una cantidad, calidad o propiedad medible. Por ejemplo; en la base de la **enigh** la edad del jefe del hogar es una característica medible, lo mismo el ingreso.
- Valor: Es el valor de una variable cuando es medida.
- Observación: Es la unidad individual de nuestro estudio, una observación contiene diferentes valores, cada uno asociado a diferentes variables. En nuestro ejemplo de la **enigh**, cada observación corresponde a la información de un hogar especifico.
- Tabla de datos: Es el conjunto de valores, asociados a una variable u observación. Una tabla de datos se caracteriza por tener cada observación en su propia fila y cada variable en su propia columna. Estas tablas de datos dentro del entorno de **R** las conocemos como data frame. En el capítulo siguiente veremos también que son objetos tipo **tibble**. Puedes imaginar una tabla de datos, precisamente como una hoja de cálculo de Excel.

2.2 Variación

La variación es la tendencia de los valores de una variable a cambiar de medida en medida. Cuando medimos una variable continua dos veces, como la temperatura, podemos obtener dos resultados distintos, cada vez que realizamos una medida obtendremos un pequeño margen de error que varia entre cada medición. Las variables categóricas también pueden variar de medida entre diferentes sujetos, cada variable tiene su propio patrón de variación, los cuales pueden revelar información interesante, la mejor forma de comenzar es visualizar la distribución de nuestras variables de interés.

Anteriormente dimos pocos detalles sobre la encuesta **enigh** que valdría la pena mencionar ahora. La encuesta está diseñada para obtener información sobre los patrones de ingresos y gastos de los hogares de manera trimestral, por ende, la uni-

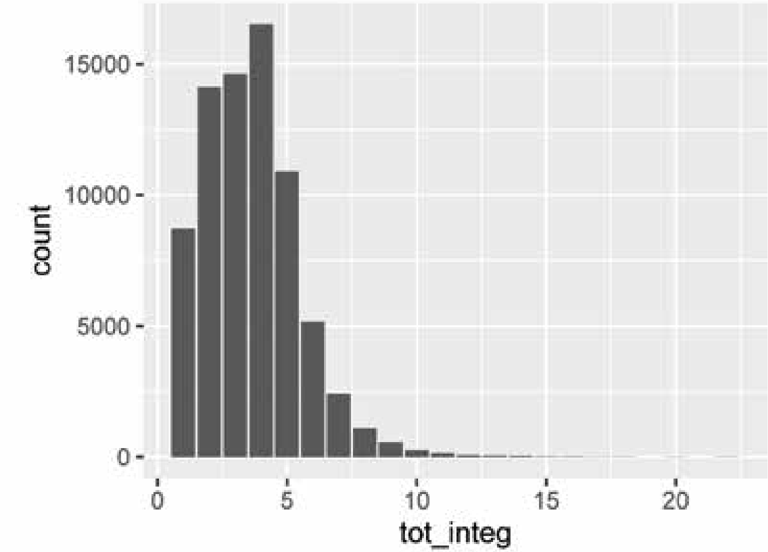
dad de observación son los hogares mexicanos. Cada observación o fila de nuestros datos representa un hogar. Además de eso, recaba información sobre las características demográficas de los integrantes del hogar, aunque en la base de datos solo se incluyen los datos del jefe del hogar, como sujeto representativo el hogar.

2.3 Distribución de los datos

Comencemos nuestro análisis realizando una distribución del total de integrantes del hogar. En este caso nuestra variable es discreta (una variable es discreta cuando puede tomar únicamente valores correspondientes a los números naturales, 1,2 ,3, etc). Para examinar esta distribución usemos una gráfica de barras:



```
ggplot(data=enigh) +
      geom_bar(mapping=aes(x=tot_integ))
```



En términos mas precisos, la figura anterior se conoce como histograma; un tipo de gráfica de barras que consiste en barras que igual anchura. El **eje x** representa clases de valores, mientras que el **eje y** representa sus frecuencias. Los histogramas nos permiten conocer la distribución de los datos, pues muestran el **centro**, la **dispersión** y nos permiten identificar valores atípicos.

En nuestro ejemplo, se observa que la mayoría de los hogares tienen entre 3 y 4 integrantes. Si la gráfica no es suficientemente clara, otra forma de calcular estos valores manualmente es con la instrucción **count()**, de esta forma obtenemos una lista con la cantidad de integrantes y la cantidad de hogares que tienen ese número de integrantes.

```
enigh %>%
     count(tot_integ)
```

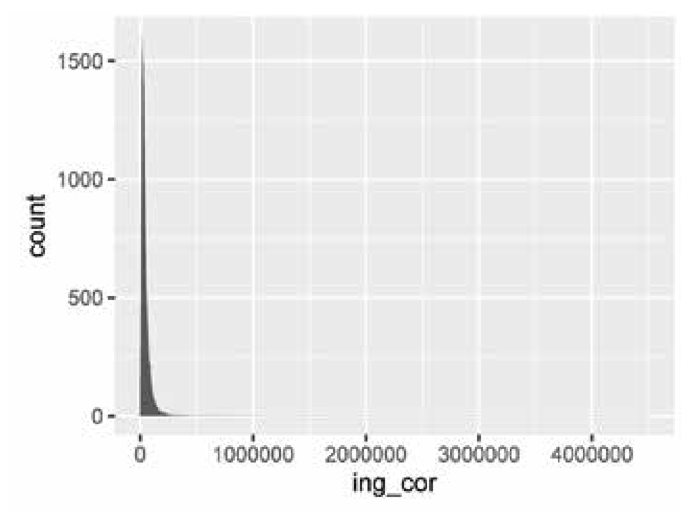
```
## # A tibble: 20 x 2
##    tot_integ    n
##        <dbl>  <int>
## 1        1     8711
```



```
## 2        2     14129
## 3        3     14624
## 4        4     16520
## 5        5     10887
## 6        6     5173
## 7        7     2413
## 8        8     1097
## 9        9     544
## 10       10    261
## 11       11    141
## 12       12    62
## 13       13    39
## 14       14    24
## 15       15    7
## 16       16    10
## 17       17    1
## 18       18    2
## 19       20    1
## 20       22    1
```

Consideremos ahora el histograma de una variable contínua (puede tomar cualquier calor), en este caso consideremos el ingreso promedio trimestral de los hogares, variable **ing_cor**. Para construir un histograma de una variable continua es necesario definir el tamaño de la clase. Recuerda que un histograma es un tipo especial de gráfica de barras.

```
ggplot(data=enigh)+
    geom_histogram(mapping = aes( x= ing_cor), binwidth = 1000)
```



Observa que en este caso hemos usado la función geométrica **geom_histogram()**, específica para efectuar un histograma. En este caso añadimos la opción **binwidth** para especificar el tamaño de la clase, es decir, agrupar los valores en clases de 1000 en 1000. Si utilizamos la forma manual para contar, podemos ver el rango de cada clase:

```
enigh %>%
     count(cut_width(ing_cor,1000,boundary = 0))
```

```
## # A tibble: 509 x 2
##    `cut_width(ing_cor, 1000, boundary = 0)`       n
##    <fct>                                       <int>
## 1  [0,1000]                                    14
## 2  (1000,2000]                                 33
## 3  (2000,3000]                                 106
## 4  (3000,4000]                                 223
## 5  (4000,5000]                                 394
## 6  (5000,6000]                                 542
## 7  (6000,7000]                                 683
## 8  (7000,8000]                                 780
## 9  (8000,9000]                                 898
## 10 (9000,1e+04]                                922
## # ... with 499 more rows
```

En este caso hemos incluido **boundary = 0** para asegurarnos que el primer rango comience en cero.

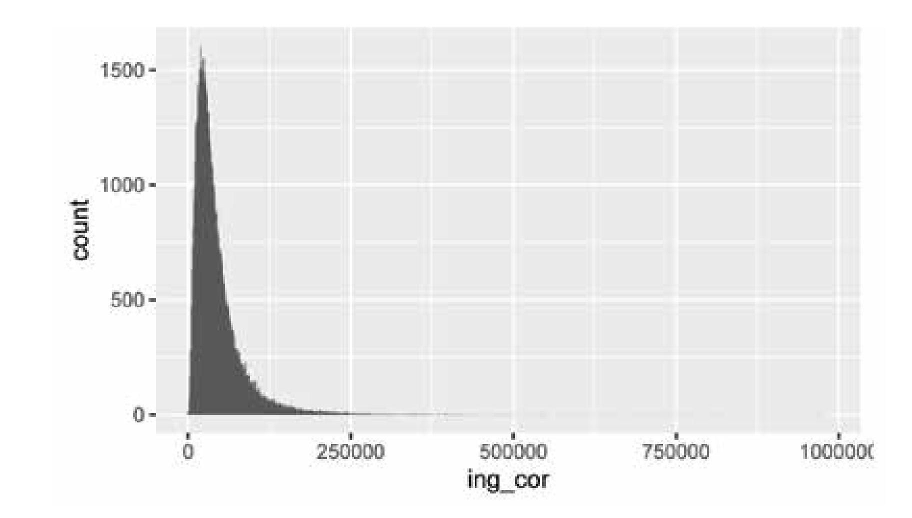
Lo primero que notamos al observar el histograma es la existencia de valores atípicos, es decir, la existencia de pocos hogares con niveles de ingreso muy altos, los cuales distorsionan la visualización del histograma.

Los datos atípicos son aquellos que se encuentran muy alejados de los demás valores de la muestra.

La visualización anterior no permite identificar el nivel de ingreso central de los hogares, por lo que debemos recortar la muestra para poder obtener una mejor visualización sin los efectos de los valores atípicos.

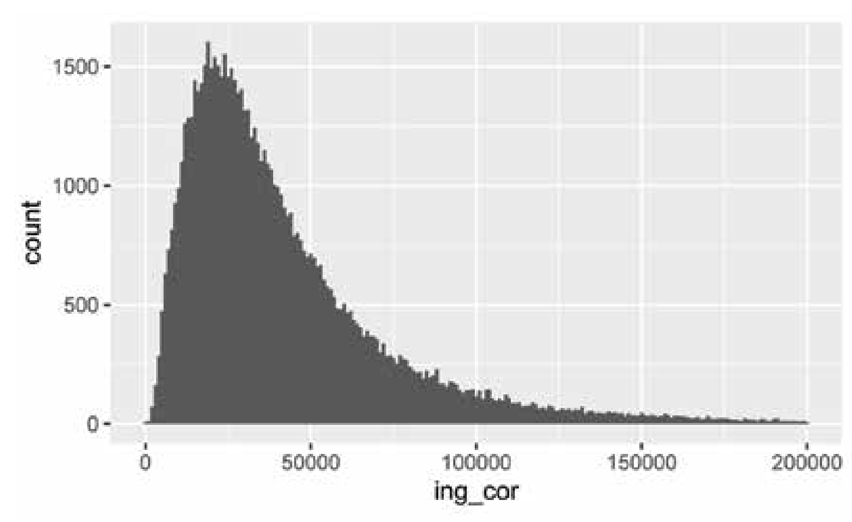
Para ello probemos excluyendo los datos mayores a un millón:

```
ggplot(data=enigh %>%
     filter(ing_cor<1000000))+
     geom_histogram(mapping = aes( x= ing_cor), binwidth = 1000)
```

Ahora probemos excluyendo los hogares con ingresos mayores a 200,000 pesos al trimestre:

```
ingresos<-enigh %>%
     filter(ing_cor<200000)
ggplot(data=ingresos)+
     geom_histogram(mapping = aes( x= ing_cor), binwidth = 1000)
```



Observa que en el primer ejemplo hemos efectuado el filtro dentro de la función **ggplot** al indicar **data=enigh %>% filter(ing_cor<1000000)** mientras qué en el segundo ejemplo, primero construimos un nuevo data frame con los datos filtrados. Ambas opciones son validas.

En nuestros datos encontraremos la variable **clase_hog**, la cual nos indica el tipo de hogar según la relación familiar de los integrantes, la variable, aunque es categórica, se encuentra codificada. Antes de hacer una gráfica es necesario reemplazarla por los valores correspondientes. En este caso no queremos generar una nueva variable, sino reemplazar los valores de una variable existente, para ello utilizaremos la función **gsub**.

(Dado que los valores se encuentran codificados recuerda consultar el descriptor de variables para conocer el significado de cada valor). Además de la función **gsub()** haremos uso del opera-

dor **$** el cual permite acceder a determinada variable en un data frame. Por ejemplo **ingresos$-clase_hog** se refiere a la variable **clase_hog** contenida en el data frame **ingresos.**

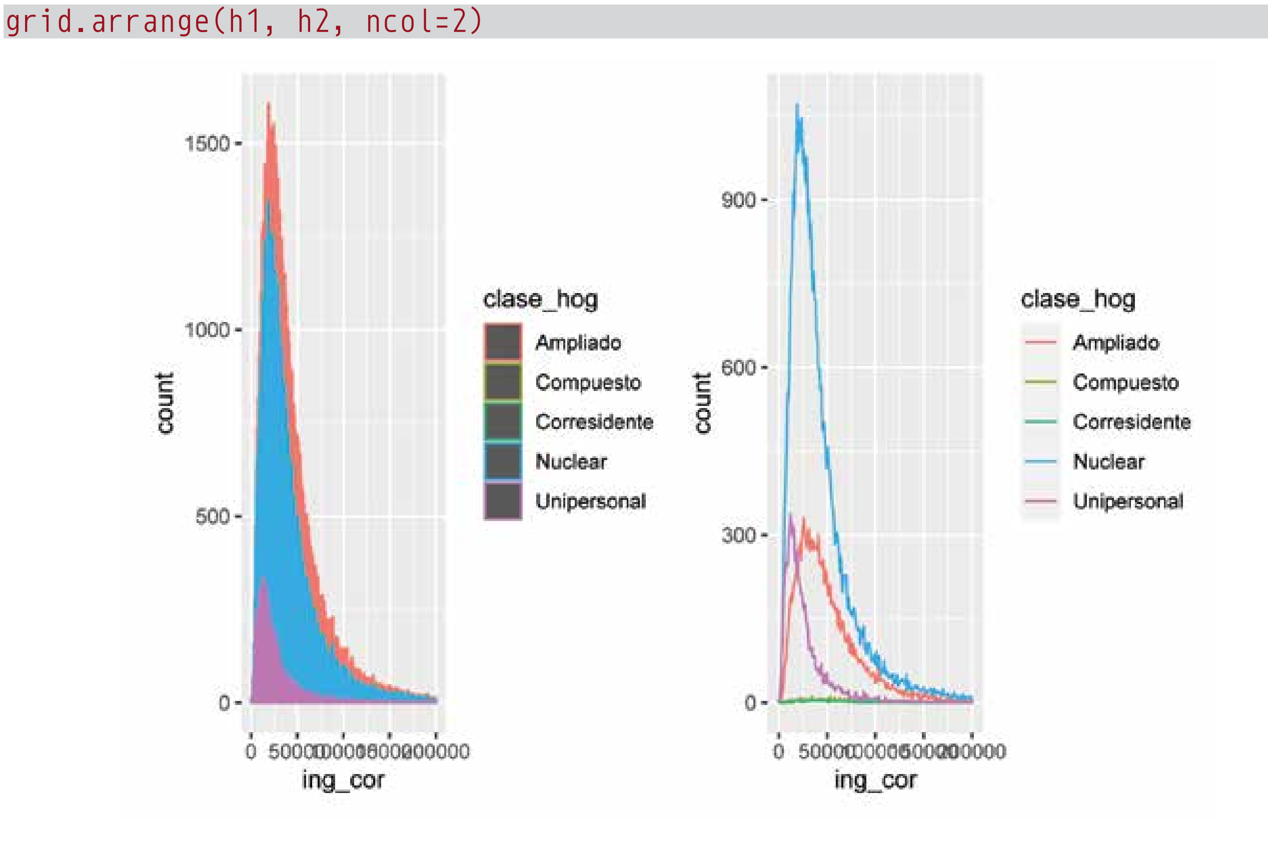
Una vez que hemos efectuado esta modificación, podemos hacer dos tipos de histogramas combinados;

```
# Reemplazamos la variable de tipo de hogar
ingresos$clase_hog ← gsub(1, "Unipersonal", ingresos$clase_hog)
ingresos$clase_hog ← gsub(2, "Nuclear", ingresos$clase_hog)
ingresos$clase_hog ← gsub(3, "Ampliado", ingresos$clase_hog)
ingresos$clase_hog ← gsub(4, "Compuesto", ingresos$clase_hog)
ingresos$clase_hog ← gsub(5, "Corresidente", ingresos$clase_hog)

h1← ggplot(data = ingresos, mapping = aes(x = ing_cor, color = clase_hog)) +
geom_histogram(binwidth = 1000)
h2← ggplot(data = ingresos, mapping = aes(x = ing_cor, color = clase_hog)) +
geom_freqpoly(binwidth = 1000)
library("gridExtra")
```

```
##
## Attaching package: 'gridExtra'
## The following object is masked from 'package:dplyr':
##
##    combine
```

```
grid.arrange(h1, h2, ncol=2)
```



Observa que hemos asignado un nombre a cada uno de los gráficos y usado la función **grid.arrange()** para pedir que estas gráficas se muestren en dos columnas. Esta función requiere la librería gridExtra para su ejecución. Nota también que la diferencia entre ambas gráficas radica en el uso de la función geométrica, ya sea geom_histogram (izquierda) o geom_freqpoly (derecha).

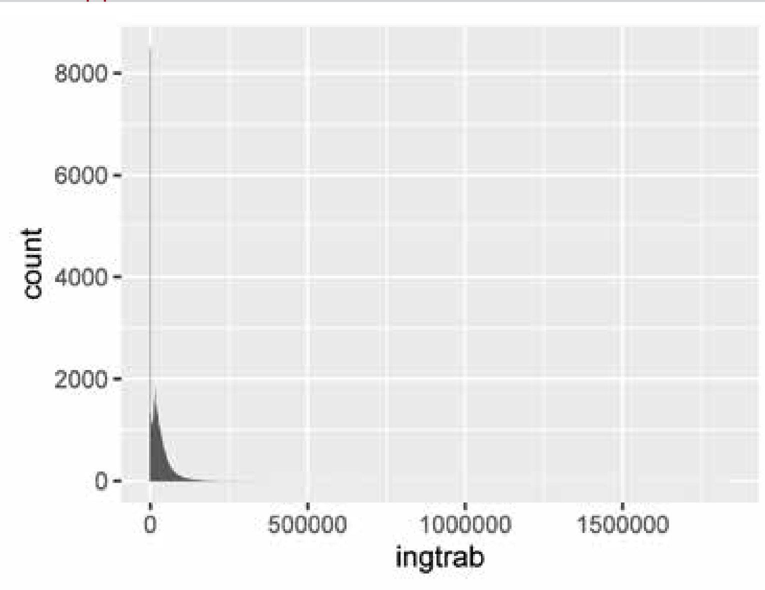
Cualquiera de las dos gráficas nos permite observar que la mayoría de los hogares son de tipo nuclear, los cuales además parece ser que son los que mayores ingresos tienen.

2.4 Valores típicos

Como seguramente has podido inferir de los ejemplos anteriores, los histogramas son muy sensibles a valores atípicos, esto hace que nuestros gráficos se vean sesgados y no podemos apreciar con facilidad los valores centrales, es decir, los valores más comunes de nuestra variable de interés.

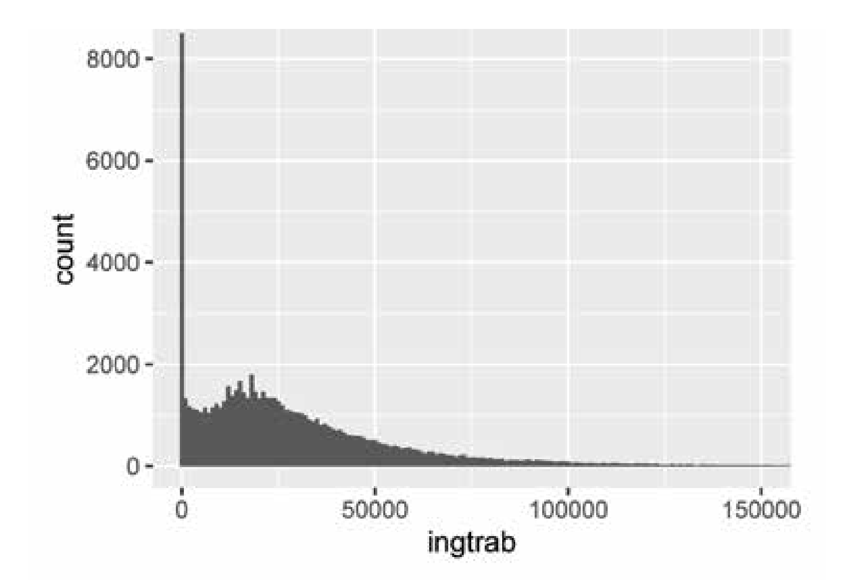
Ahora hagamos un histograma con la variable ingtrab la cual indica el promedio trimestral de ingreso por trabajo de los hogares.

```
ggplot(data=enigh)+
     geom_histogram(mapping = aes( x= ingtrab), binwidth = 1000)
```



La visualización por si misma nos dice poco, de nuevo tenemos valores atípicos que dificultan encontrar los valores centrales. Probemos ahora realizando un “acercamiento” a los valores entre 0 y 200,000 pesos, para ello utilizamos la función **coord_cartesian**, en conjunto con la opción **xlim** para indicar el rango de valores de x que deseamos observar en el histograma:

```
ggplot(enigh) +
geom_histogram(mapping = aes(x = ingtrab), binwidth = 1000) +
coord_cartesian(xlim = c(0, 150000))
```

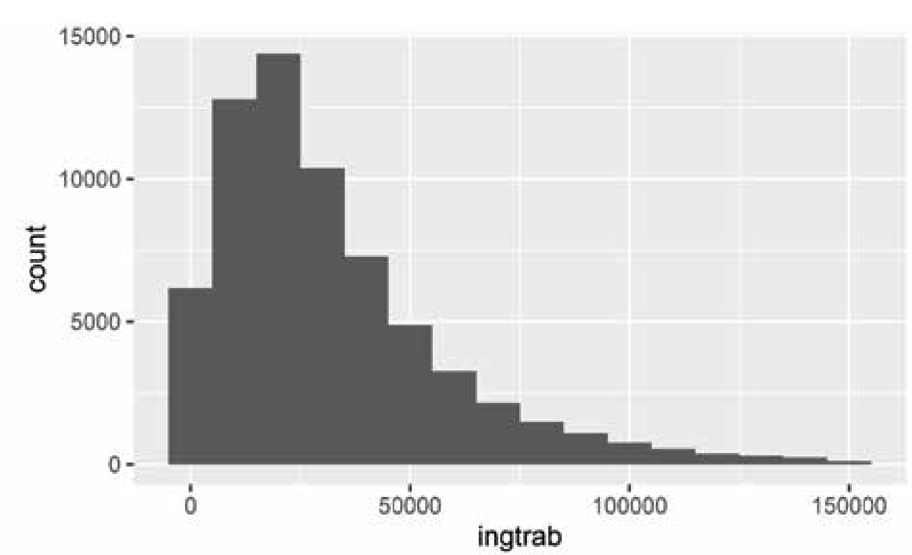


La opción “coord_cartesian” permite recortar el eje x con el argumento xlim, también podemos recortar el eje y con el argumento ylim en caso de que requiera.

¿Notas algo incongruente en la gráfica? existe un valor que se repite muchas veces, mas que ningún otro. Este valor es el cero. Sí revisamos nuestra variable, notaremos que existen hogares que declaran cero ingreso por trabajo. Probemos construir un nuevo **data frame**, donde no se consideren los valores de ingreso cero, pero tampoco aquellos mayores a 150,000:

```
ingresos_trabajo <- enigh %>%
filter(ingtrab >0 & ingtrab < 150000)

ggplot(ingresos_trabajo) +
geom_histogram(mapping = aes(x = ingtrab), binwidth = 10000)
```



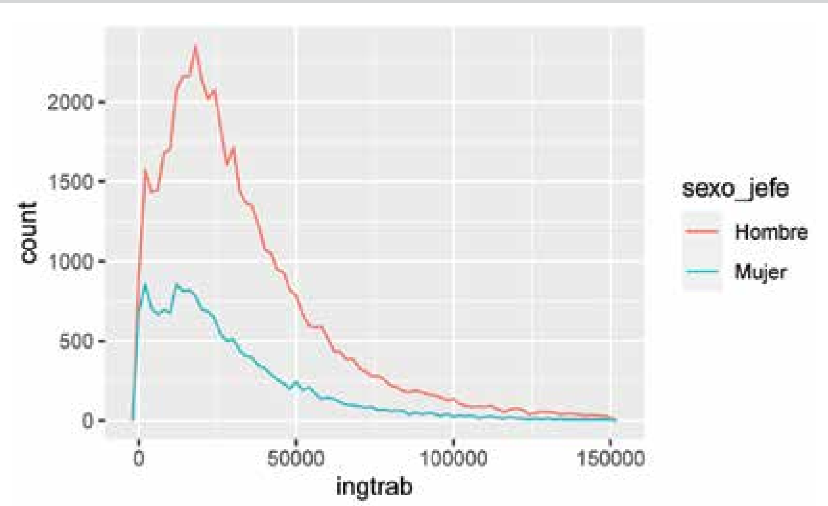
Finalmente podemos decir que los valores centrales de los ingresos por trabajo son alrededor de 30,000 pesos al trimestre.

Ahora, combinemos esta información con el sexo del jefe del hogar para observar si existen diferencias entre hogares con jefatura masculina y femenina. Para ello observemos primero que la variable sexo_jefe contiene valores númericos de 1 y 2. Para poder efectuar la separación está variable debe ser categorica (es decir que 1 defina no un numero si no una categoría, lo mismo para 2.) Entonces es necesario transformar esta variable usando **gsub()** como lo hicimos anteriormente

```
ingresos_trabajo$sexo_jefe<-gsub(1,"Hombre", ingresos_trabajo$sexo_jefe)
ingresos_trabajo$sexo_jefe<-gsub(2,"Mujer", ingresos_trabajo$sexo_jefe)
```

Una vez efectuado el cambio, tenemos que;

```
ggplot(data = ingresos_trabajo, mapping = aes(x = ingtrab, color = sexo_
jefe))+
geom_freqpoly(binwidth = 2000)
```



¿Qué podemos concluir de la gráfica anterior? Que los hogares con jefatura femenina, tienen menores ingresos que los hogares de jaftura femenina.

2.5 Valores perdidos

Anteriormente observamos que existen valores atípicos en nuestra muestra y para poder analizar un histograma fue necesario filtrar nuestros datos o recortar el eje del gráfico. En ocasiones será necesario realizar esta acción para diferentes variables, generar un nuevo **data frame** cada vez que realizamos un cambio es poco eficiente. Por ello es aconsejable realizar estos cambios sobre un mismo conjunto de datos, la forma más fácil de tratar con datos atípicos es eliminarlos, sin embargo, puede ser posible que no deseemos eliminar todas estas observaciones sino más bien solo algunos casos de diferentes variables. Todo dependerá del análisis que deseemos realizar.

Anteriormente vimos que el ingreso corriente contiene ceros, ya sea por no respuesta o porque en realidad un hogar declaro tener cero ingreso, también observamos que existen pocos hogares con niveles de ingreso muy altos.

A continuación, aprenderemos como reemplazar estos valores y los valores perdidos. Como su nombre lo indica, un valor perdido equivale a una valor omitido o inexistente por lo tanto **R** ignora estos casos y se visualizan con una etiqueta **"NA"** siglas del ingles "No available".

Antes de continuar veamos un resumen del data frame **enigh**, en la variable ing_cor.

```
summary(enigh$ing_cor)
##    Min.  1st Qu.    Median     Mean       3rd  Qu.    Max.
##       0    20345     33573     46044     55196     4501830
```

Esta indicación nos muestra las estadísticas generales para la variable solicitada. Observa que nos dice que el valor mínimo de la variable es cero y el máximo es 4501830.

Si ejecutamos el siguiente código reemplazaremos la base enigh anterior, por una donde los hogares en los que el ingreso corriente es cero o mayor a 200000, se les ha asignado un NA.

```
enigh <- enigh %>%
mutate(ing_cor = ifelse(ing_cor==0 | ing_cor>200000 , NA, ing_cor))

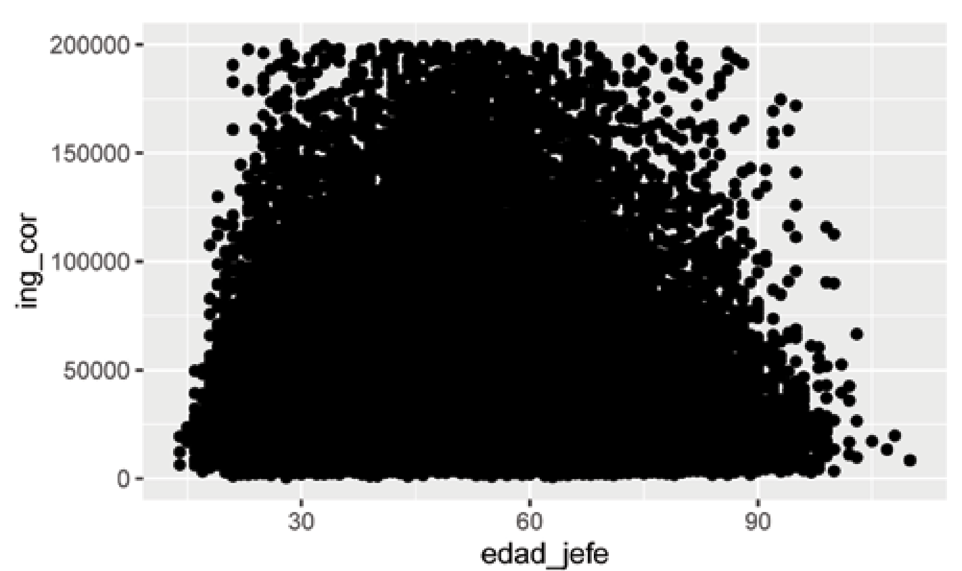
summary(enigh$ing_cor)
##    Min.     1st Qu.    Median     Mean      3rd Qu.    Max.       NA's
##    381.3    20183.7    33180.8    42215.6   53971.2    199967.2   937
```

La indicación **ifelse(ing_cor==0 | ing_cor>200000** , NA, ing_cor) instruye a que primero se haga una comparación de la variable ing_cor. Si su valor es cero o mayor a 200000, se cambie por NA, de lo contrario se asigne el valor actual.

Observa que ahora el resumen de la variable arroja la existencia de 937 NA, resultantes del proceso efectuado.

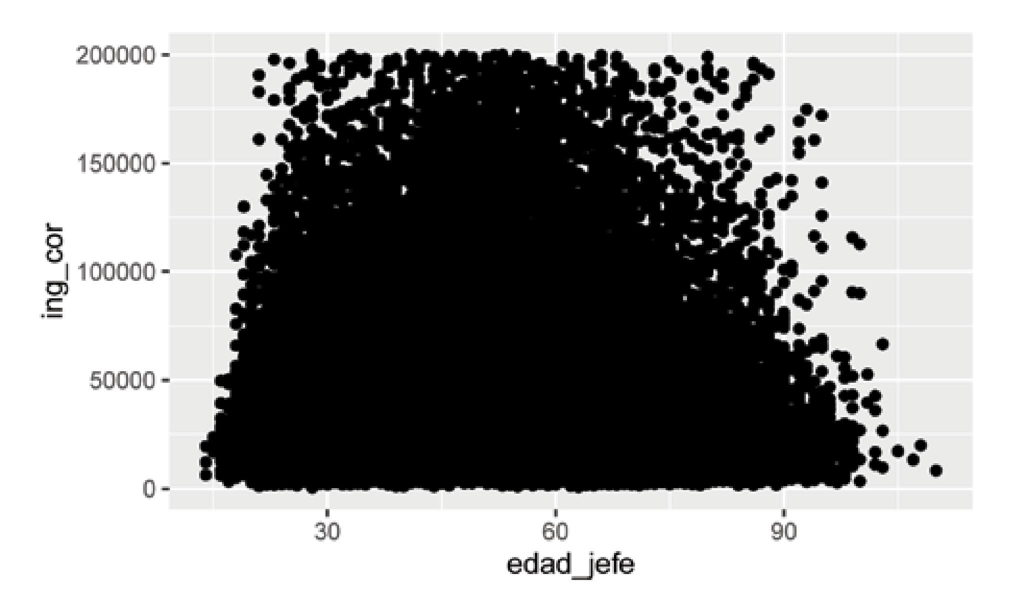
Ahora si realizamos un gráfico de dispersión entre el ingreso corriente del hogar y la edad del jefe del hogar, notaremos una advertencia sobre la cantidad de valores perdidos que se están ignorando:

```
ggplot(data = enigh, mapping = aes(x = edad_jefe, y = ing_cor )) +
geom_point()
## Warning: Removed 937 rows containing missing values (geom_point).
```

Para evitar estas advertencias es necesario utilizar la instrucción na.rm = TRUE, la cual indica que se remuevan los NA presentes en la base de datos al momento de hacer la gráfica.

```
ggplot(data = enigh, mapping = aes(x = edad_jefe, y = ing_cor)) +
geom_point(na.rm = TRUE)
```



2.6 Co-variación

La co-variación se refiere al comportamiento entre variables, es decir, la tendencia de los valores de dos variables o más a cambiar. La mejor manera de entender este concepto es visualizar la relación entre dos variables. Estas relaciones dependerán de los tipos de variables que estemos trabajando. En general se identifican tres casos; - Una variable categórica con una continua - Dos variables categóricas - Dos variables continuas.

2.6.1 Categórica con continua

Antes de continuar, limpiemos un poco nuestro ambiente de trabajo

```
remove (ingresos, ingresos_trabajo, sexo, tabla_entidad, entidades, edades,
h1, h2, clases_hogar)
```

Ahora, realicemos los cambios necesarios a nuestro data frame enigh. Recordemos que es mejor reemplazar los valores en nuestra misma base de datos que generar nuevos conjuntos. En este ejemplo usaremos las variables sexo_jefe y clase_hog. Observa que ambas identifican a través de un numero tanto el sexo del jefe como el tipo de hogar

```
summary(enigh$sexo_jefe)
##    Min.    1st Qu.    Median    Mean    3rd Qu.    Max.
##    1.000   1.000      1.000     1.274        2.000      2.000
```

```
summary(enigh$clase_hog)
##    Min.    1st Qu.    Median    Mean    3rd Qu.    Max.
##    1.000   2.000      2.000     2.148   3.000      5.000
```

Este resumen nos indica que **R** está considerando las variables como numéricas y no como categóricas. Hagamos lo mismo que hicimos la función **gsub**, para definir categorías mas que valores numéricos.

```
enigh$sexo_jefe<-gsub(1,"Hombre", enigh$sexo_jefe)
enigh$sexo_jefe<-gsub(2,"Mujer", enigh$sexo_jefe)
enigh$clase_hog<-gsub(1, "Unipersonal", enigh$clase_hog)
enigh$clase_hog<-gsub(2, "Nuclear", enigh$clase_hog)
enigh$clase_hog<-gsub(3, "Ampliado", enigh$clase_hog)
enigh$clase_hog<-gsub(4, "Compuesto", enigh$clase_hog)
enigh$clase_hog<-gsub(5, "Corresidente", enigh$clase_hog)
```

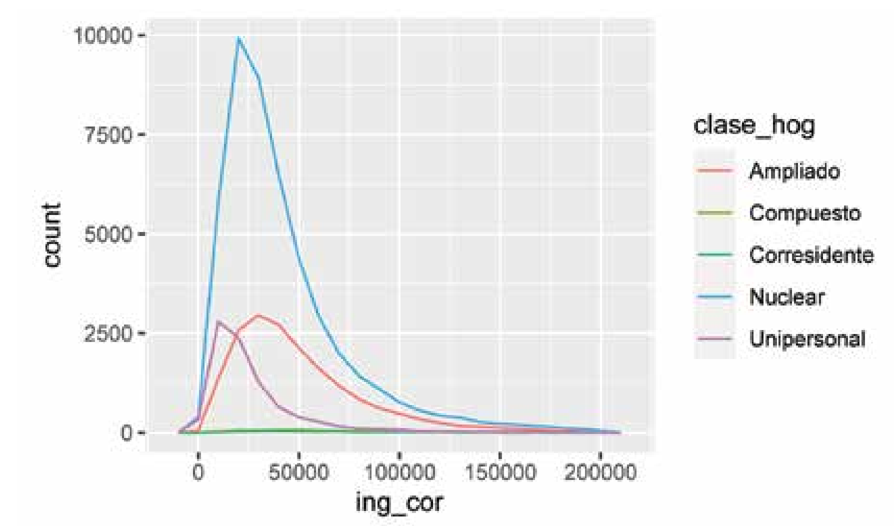
Con estos cambios, nuestras variables ahora son de tipo carácter.

```
summary(enigh$sexo_jefe)
##    Length    Class        Mode
##    74647     character  character
```

```
summary(enigh$clase_hog)
##    Length    Class        Mode
##    74647     character  character
```

Tal como lo hicimos anteriormente, analizamos el ingreso de los hogares según la clase de hogar y sexo, sin embargo, en los histogramas, dado que estamos analizando la frecuencia, no podemos apreciar de manera clara las diferencias entre el ingreso según la clase del hogar. Lo que observamos es que hay muchos hogares de tipo nuclear y muy pocos hogares del tipo corresidente y compuesto:

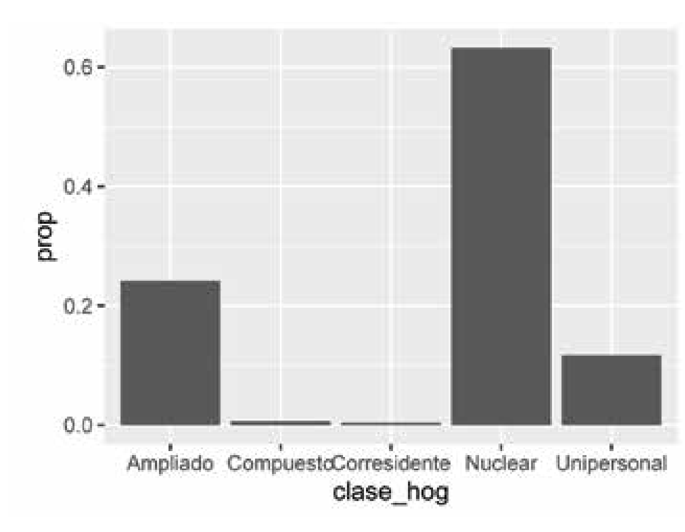

```
ggplot(data = na.omit(enigh), mapping = aes(x = ing_cor)) +
geom_freqpoly(mapping = aes(color = clase_hog), binwidth = 10000)
```



En efecto, existen diferencias en el tamaño de cada grupo de clase de hogares, algunos grupos son mucho más pequeños que otros. Es difícil ver las diferencias en los histogramas cuando el tamaño de cada grupo es muy diferente.

Para visualizar estas diferencias con mayor claridad, elaboremos una gráfica de barras, con proporciones.

```
ggplot(data=enigh)+
    geom_bar(mapping = aes(x=clase_hog, y=..prop.., group=1))
```



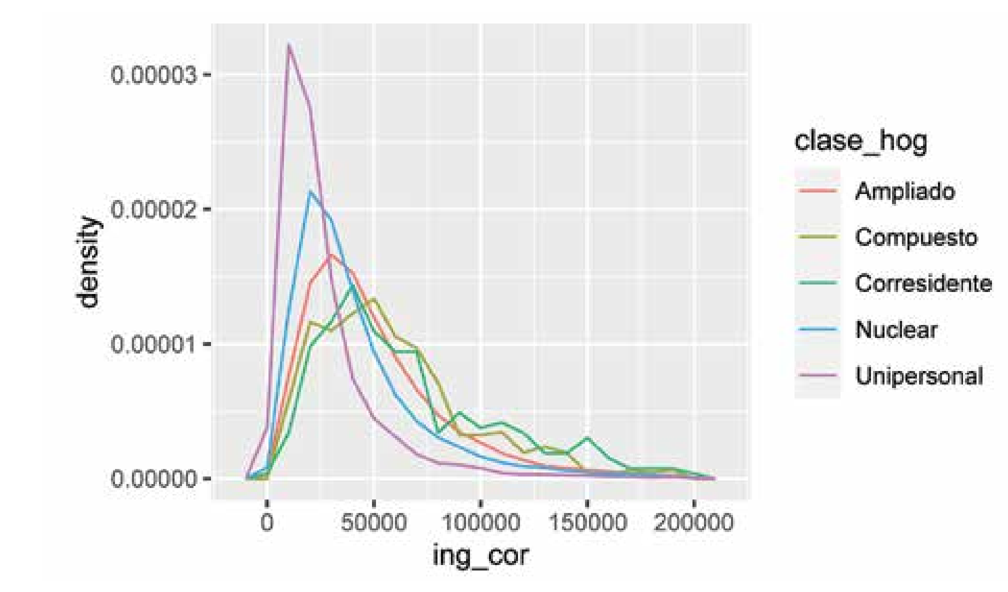
Para poder realizar una mejor comparación si realizamos un histograma, pero en lugar de mostrar la frecuencia en el eje vertical, mostramos la densidad, la cual indica el frecuencia estandarizada, de tal manera que el área bajo cada curva es igual a uno. Esto lo hacemos con la indicación **y = ..density..**

Si realizamos el histograma de esta manera, obtenemos una visualización mejor, por ende, podemos describir mejor nuestros datos.

```
ggplot(
data = na.omit(enigh),
```

```
mapping = aes(x = ing_cor, y = ..density..)
) +
geom_freqpoly(mapping = aes(color = clase_hog), binwidth = 10000)
```

Observa que en el histograma normal (sólo de frecuencias), no se incluyó ninguna variable en el eje y. En este caso sí lo hicimos.

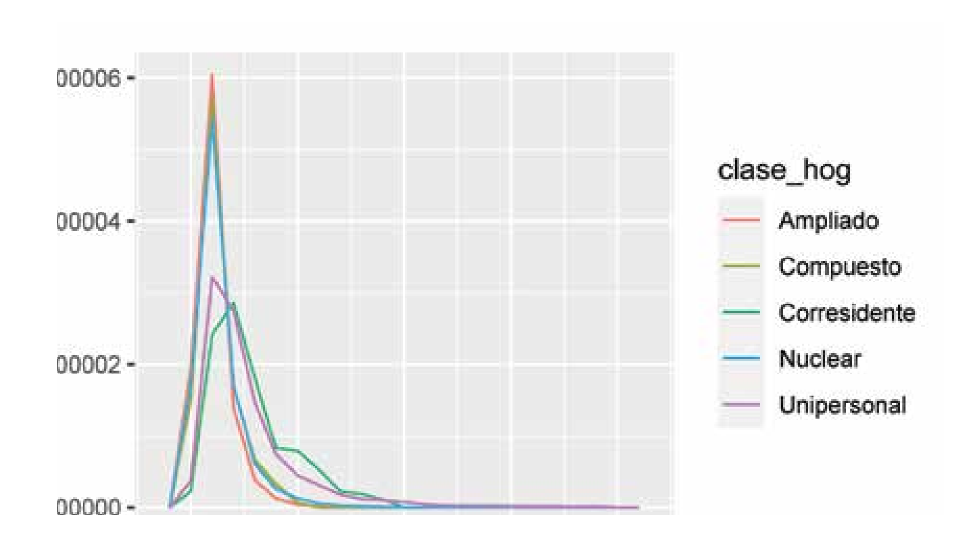


Existe otro factor que puede alterar la distribución del ingreso, que es el tamaño del hogar. Hogares con mayor número de habitantes tendrán mayor nivel de ingreso, no porque posean mayor riqueza, sino porque un mayor número de personas participarán en actividades remuneradas y por ende aumentará el ingreso total del hogar. Para corregir esta situación necesitamos analizar el ingreso per cápita, de esta forma tendremos un mejor indicador del nivel de ingreso de cada hogar. Para ello debemos calcular el ingreso per cápita, haciendo uso de los elementos que hemos aprendido con anterioridad.

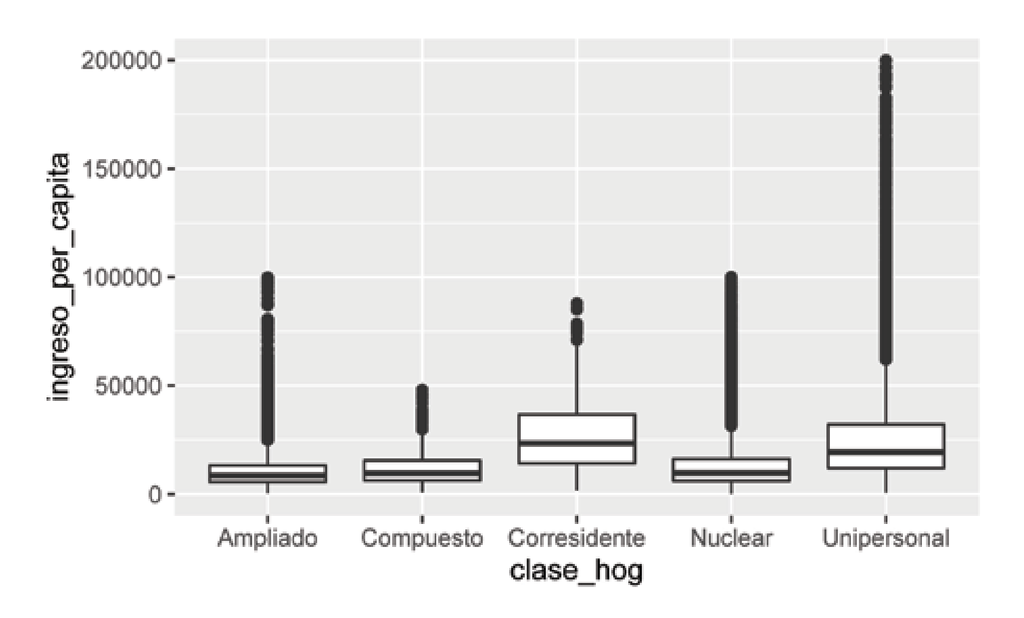
```
enigh <- enigh %>%
mutate(ingreso_per_capita = ing_cor/tot_integ )
```

```
ggplot(
data = na.omit(enigh),
mapping = aes(x = ingreso_per_capita, y = ..density..)
) +
geom_freqpoly(mapping = aes(color = clase_hog), binwidth = 10000)
```

Finalmente, una manera sencilla de resumir información entre categorías es con un diagrama de caja. Con el siguiente diagrama de caja observamos el comportamiento del ingreso per cápita por clase de hogar:




```
ggplot(data = na.omit(enigh), mapping = aes(x = clase_hog, y = ingreso_per_
capita)) +
geom_boxplot()
```



Recuerda que un diagrama de caja muestra los máximos, mínimos, promedios y demás cuartiles de una variable para cada categoría. En este caso, el mayor ingreso máximo se observa en los hogares de tipo unipersonal y los hogares con mayor promedio de ingreso son los corresidentes. Podemos corroborar esto haciendo un tabulado

```
summarise ( group_by(na.omit(enigh), clase_hog),
mean(ingreso_per_capita), max(ingreso_per_capita), min(ingreso_per_capita),
.groups = "drop")
```

```
## # A tibble: 5 x 4
##   clase_hog `mean(ingreso_per_cap~ `max(ingreso_per_cap~ `min(ingreso_per_
```

```
cap~
##    <chr>              <dbl>      <dbl>      <dbl>
## 1  Ampliado          10842.     99752.     378.
## 2  Compuesto         12417.     48094.     893.
## 3  Corresiden~       28174.     87807.     1691.
## 4  Nuclear           13170.     99967.     127.
## 5  Unipersonal       26956.     199860.    681.
```

Observa que en varios casos hemos usado la indicación `na.omit(enigh)` la cual indica que se omitan los valores NA presentes en la base, pues recuerda que hay algunas observaciones con valor nulo.

```
summary(enigh$educa_jefe)
```

```
##    Length     Class          Mode
##    74647      character  character
```

Se trata de una variable de tipo carácter. Si observamos los primero 5 valores que toma esta variable, tenemos que

```
enigh$educa_jefe[1:5]
```

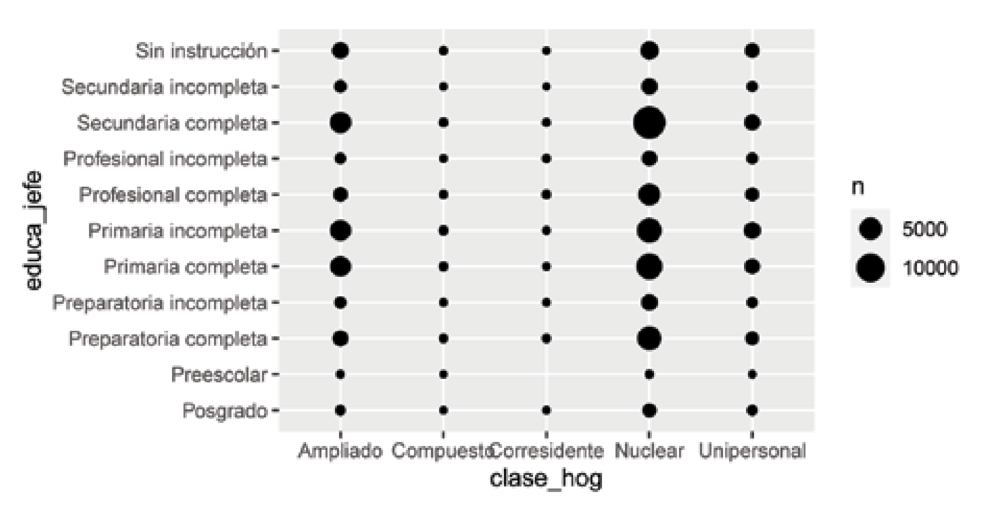
[1] "04" "11" "10" "08" "04"

Para tener mayor claridad sobre que significa que un jefe tenga educación 01, transformaremos cada carácter en una definición, más fácil de entender. Esto lo haremos considerando las nomenclaturas que hace INEGI, la institución que efectúa esta encuesta. Para ello podemos hacer, uso de **gsub()**, como lo hicimos anteriormente. Probemos ahora mutando el data frame **enigh**, simplemente para practicar esta instrucción.

```
enigh <- enigh %>%
   mutate(educa_jefe =
      ifelse(educa_jefe=="01" , "Sin instrucción" ,
      ifelse(educa_jefe=="02","Preescolar" ,
      ifelse(educa_jefe=="03","Primaria incompleta" ,
      ifelse(educa_jefe=="04", "Primaria completa" ,
      ifelse(educa_jefe=="05","Secundaria incompleta" ,
      ifelse(educa_jefe=="06", "Secundaria completa" ,
      ifelse(educa_jefe=="07", "Preparatoria incompleta" ,
      ifelse(educa_jefe=="08", "Preparatoria completa" ,
      ifelse(educa_jefe=="09", "Profesional incompleta" ,
      ifelse(educa_jefe=="10", "Profesional completa" , "Posgrado"
      ))))))))))
```


Ahora contamos el número de hogares en cada grupo. Una forma sencilla de hacerlo es a través de una gráfica de círculos, donde cada circulo representa cuantos hogares existen en cada combinación de variables.

```
ggplot(data = enigh) +
geom_count(mapping = aes(x = clase_hog, y = educa_jefe))
```



Otra manera de ver el conteo es de forma tabular, usando la función **count()**:

```
enigh %>%
count(educa_jefe, clase_hog)
```

```
## # A tibble: 54 x 3
##    educa_jefe             clase_hog        n
##    <chr>                  <chr>            <int>
## 1  Posgrado               Ampliado         182
## 2  Posgrado               Compuesto        3
## 3  Posgrado               Corresidente     9
## 4  Posgrado               Nuclear          864
## 5  Posgrado               Unipersonal      204
## 6  Preescolar             Ampliado         25
## 7  Preescolar             Compuesto        1
## 8  Preescolar             Nuclear          37
## 9  Preescolar             Unipersonal      14
## 10 Preparatoria completa Ampliado 1488
## # ... with 44 more rows
```

Finalmente, una forma más atractiva de ver el tamaño de los grupos es usando un mapa de calor. Este se puede generar con la función de geometría **geom_tile()** :

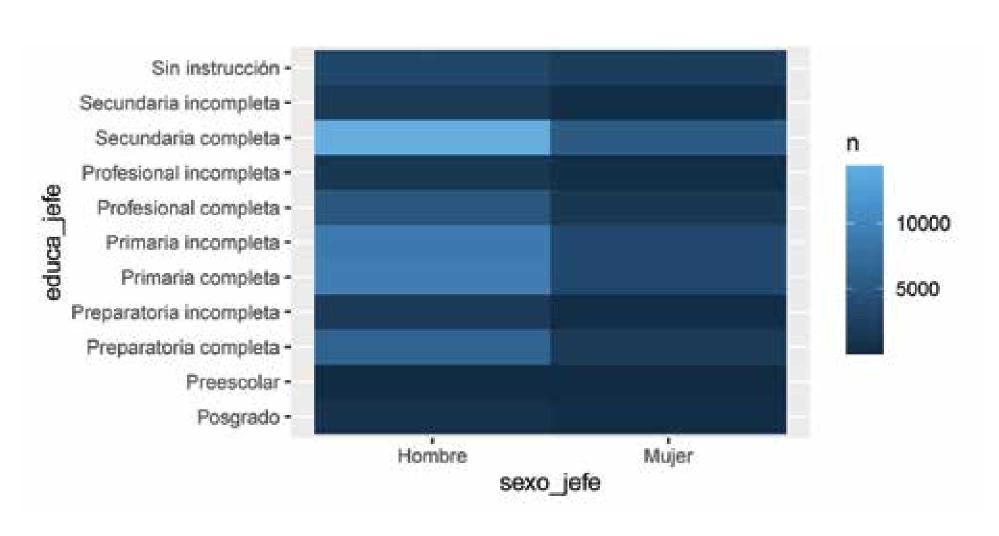
```
enigh %>%
     count(educa_jefe , clase_hog) %>%
     ggplot(mapping = aes(x = clase_hog , y = educa_jefe)) +
     geom_tile(mapping = aes(fill = n))
```

Observa que primero hemos contado los datos en cada categoría, luego hemos efectuado el mapa de calor. La función de estética **fill=n** indica que el relleno de la gráfica considere el número de observaciones en cada categoría. La barra de color en la derecha indica que entre mas claro sea el azul, mayor cantidad de hogares habrá en esa categoría. En este caso existen mas hogares de tipo nuclear con secundaria completa. En la gráfica anterior podemos hacer **fill = -n** para que la intensidad tenga el significado opuesto. En ese caso entre mas obscuro mas cantidad de hogares. Pruébalo por ti mismo.

Podemos también analizar los hogares por educación y sexo. Para ello cambiamos la indicación del **eje x** del código anterior.

```
enigh %>%
count(educa_jefe , sexo_jefe) %>%
ggplot(mapping = aes(x = sexo_jefe , y = educa_jefe)) +
geom_tile(mapping = aes(fill = n))
```

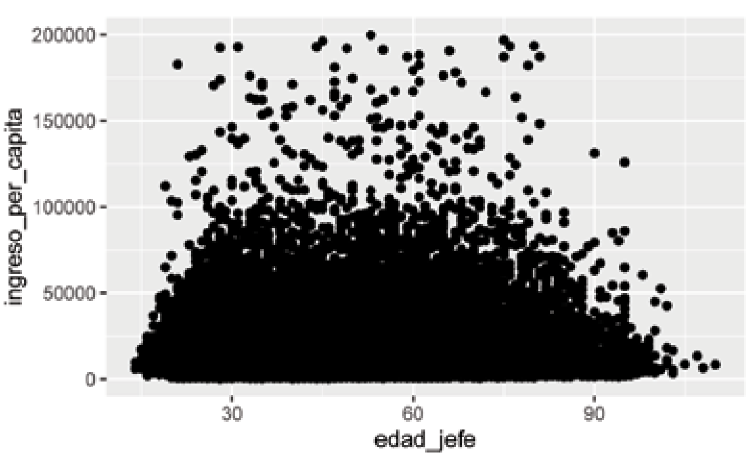
En esta nueva desagregación, el mayor número de hogares se ubica en hombres, con secundaria completa.



2.6.3 Dos variables continuas

Una de las mejores gráficas para visualizar la relación entre dos variables continuas, es a través de una gráfica de dispersión, la cual ya hemos utilizado con anterioridad. En esta gráfica podemos observar la asociación entre variables. Consideremos el ejemplo del ingreso per cápita del hogar y la edad del jefe del hogar:

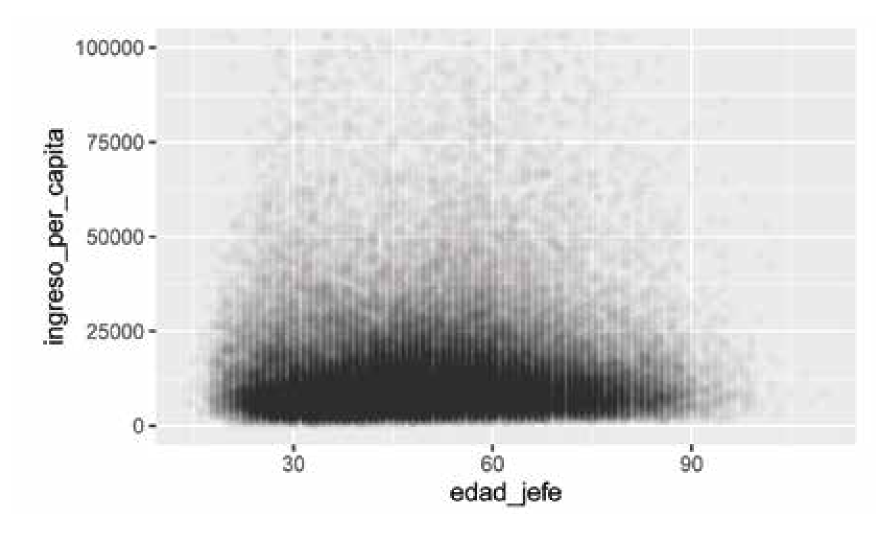
```
ggplot(data =na.omit(enigh)) +
geom_point(mapping = aes(x = edad_jefe, y = ingreso_per_capita))
```



Podemos agregar transparencia para evitar que nuestra visualización se sature (debido al número de observaciones), la transparencia nos permite observar mejor en donde se concentran los valores, se intuye que el ingreso aumenta hasta alrededor de los 60 años y comienza a descender.

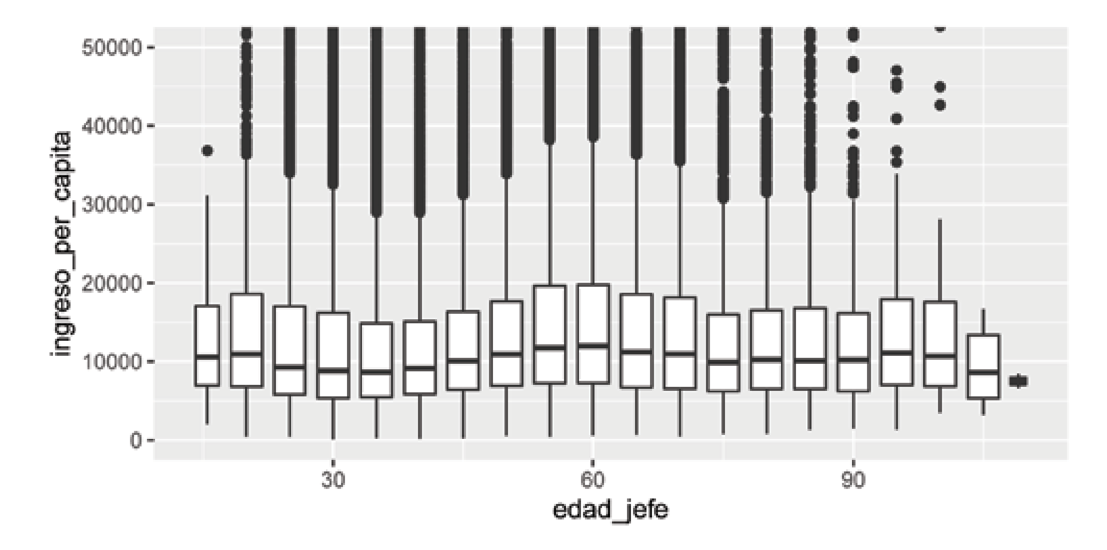
```
ggplot(data = enigh) +
geom_point(
mapping = aes(x = edad_jefe, y = ingreso_per_capita),
alpha = 1 / 100
)+
coord_cartesian(ylim = c(0, 100000))
```

```
## Warning: Removed 937 rows containing missing values (geom_point).
```



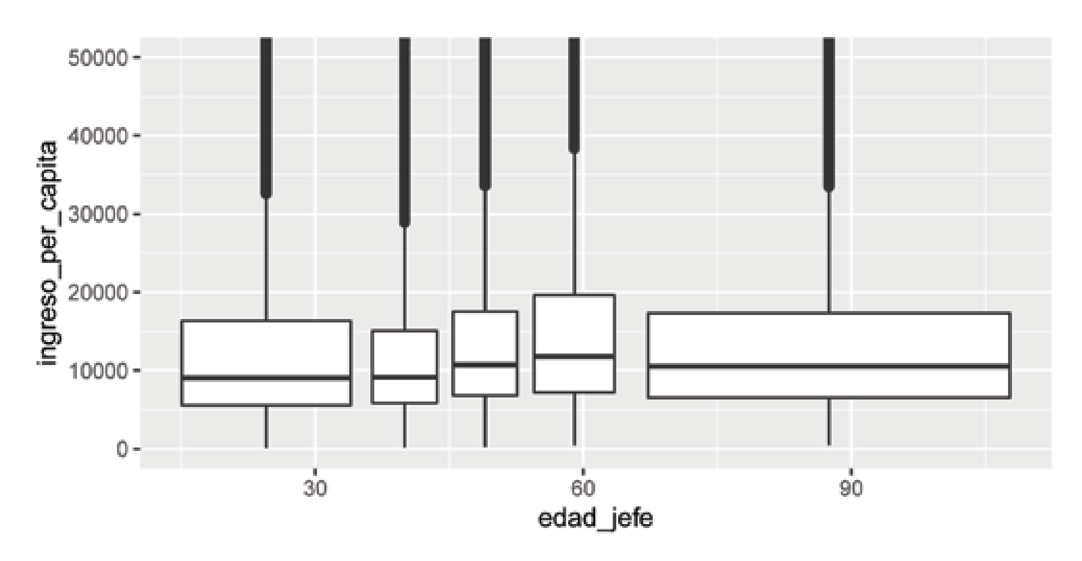
Finalmente, si aún no podemos observar un patrón claro en nuestras variables, podemos agrupar una variable continua para observar cambios entre grupos, como si fuera una variable categórica, por ejemplo, realicemos un diagrama de caja, pero agrupando a los hogares en grupos de edad de 5 en 5 años, la instrucción **cut_width()** nos permite indicar el tamaño de cada grupo:

```
ggplot(data = enigh, mapping = aes(x = edad_jefe, y = ingreso_per_capita)) +
     geom_boxplot(mapping = aes(group = cut_width(edad_jefe, 5)),na.rm =TRUE)
+
     coord_cartesian(ylim = c(0, 50000))
```



En cambio, si deseamos que cada grupo tenga aproximadamente el mismo número de observaciones, podemos utilizar la instrucción cut_number(), para indicar el número de clases o grupos que necesitamos y los agrupa de manera automática:

```
ggplot(data = na.omit(enigh), mapping = aes(x = edad_jefe, y = ingreso_per_
capita)) +
     geom_boxplot(mapping = aes(group = cut_number(edad_jefe, 5)))+
     coord_cartesian(ylim = c(0, 50000))
```



Esperamos que hayas notado que en algunos casos **R** te ha indicado un warning como el siguiente; *Warning: Removed 937* rows containing… En algunos casos lo hemos omitido en el código con fines de aprendizaje. Recuerda que si esto te apareció es porque existen valor nulos o NA, los cuales al efectuar la gráfica **R** ha omito de manera automática. Y **R** solo quiere informarrte que existe esa inconsistencia. En este capitulo hemos visto dos formas de lidiar con ellos, la primera con el uso de **na.rm =TRUE** dentro de la la geometría de la función y la segunda haciendo **data=na.omit()**

Queremos cerrar este capítulo haciendo una observación importante. La decisión sobre la permanencia o no de valores atípicos en un conjunto de datos se relaciona con el objetivo final del análisis que se efectúa. En este caso hemos convertido esos valores atípicos en NA y los hemos omitido. Si estuviéramos, por ejemplo, haciendo un análisis formal de desigualdad en el ingreso, eliminar los ceros, o los valores muy grandes sería incorrecto, porque un análisis de desigualdad busca comparar aquellos que tienen mucho, contra aquellos que tienen poco. En consecuencia, que hacer con los valores atípicos es decisión del analista y de los objetivos que persigue en su investigación.

3 Actividades

1. Utiliza el data frame enigh para crear un nuevo conjunto de datos con las siguientes características:

• Hogares de jefatura femenina menores de 30 años

• Hogares de jefatura masculina menores de 30 años

• Hogares de unipersonales menores de 30 años (los hogares unipersonales corresponden a la clave 1 de la variable "clase_hog")

2. Genera un nuevo data frame con las siguientes variables: ubica_geo, edad_jefe, tot_integ, gasto_mon, alimentos, vesti_calz, vivienda, limpieza, salud, transporte, educa_espa, personales y transf_gas. Donde las variables son alimentos, vesti_calz, vivienda, limpieza, salud, transporte, educa_espa, personales y transf_gas son rubros de gasto que componen al gasto monetario (gasto_mon). Con este nuevo conjunto de datos, calcula lo siguiente:

• El porcentaje que representa cada rubro respecto al gasto monetario total. Da el nombre de gastos a este data frame.

• Encuentra gráficamente a qué edad del jefe del hogar se alcanza el máximo gasto per cápita en salud

3. Haciendo uso del data frame gastos generado en 2, obtén una tabla por entidad federativa donde se muestren tres cosas; A) el gasto promedio per cápita en salud, B) el promedio del gasto en salud como porcentaje del gasto monetario y C) el gasto monetario promedio. Con la información obtenida, ¿Cuál es el estado con mayor gasto per cápita en salud?

4. Realiza un histograma del ingreso monetario de los hogares

• Identifica valores atípicos

• Reemplaza los valores atípicos como valores perdidos

5. Calcula el gasto monetario per cápita

• Analiza la distribución del gasto per cápita de los hogares por sexo del jefe del hogar.

Capítulo 5

Importación de datos

Capítulo 5 | Importación de datos

1 Introducción

En capítulos anteriores hemos trabajado usando bases de datos que se encuentran en formato Excel, las cuales hemos cargado al entorno de **R**, gracias al uso de funciones de la librería **readxl**. Esto ha sido de mucha ayuda. Sin embargo, en ocasiones requerimos utilizar bases de datos qué por su tamaño, se encuentra en formatos distintos a Excel. El más común de estos formatos es csv. En la primera parte de este capítulo nos enfocaremos en entender el proceso de importación de datos usando esta librería. Para ello, comenzaremos con los elementos básicos y después pondremos a prueba lo aprendido cargando diferentes bases. En el segundo apartado, conoceremos las diferencias que hay entre dos objetos para el manejo de datos en **R**, nos referimos a los objetos de tipo **tibble** y los **data frame**.

2 Primeros pasos para importar con readr

Necesitaremos la librería **readr**, la cual se encuentra dentro de tidyverse. Por lo cual es necesario que instales esta librería, si es que aún no la tienes. Si ya la tienes, únicamente debes decirle a **R** que requerimos su uso.

```
library(tidyverse)
```

La tarea principal de las funciones contenidas en esta librería gira en torno a convertir archivos de texto en **data frame**.

Antes de pasar al uso de la librería para cargar datos a R, veamos cual es la lógica bajo la cual trabaja

la función principal read_csv, para transformar la lectura de un archivo de texto en un formato de base de datos. Para ello probemos los siguientes códigos

```
read_csv("sexo, edad, ingreso
           1,40,5000
           2, 25, 2300
           1, 15, 4600")
```

```
## # A tibble: 3 x 3
##    sexo  edad ingreso
##  <dbl>  <dbl>          <dbl>
## 1  1     40         5000
## 2  2     25         2300
## 3  1     15         4600
```

Observa que los tres elementos que se encuentran en la primera línea corresponden al nombre de las columnas **(variables), después de eso cada nueva línea (observación) se compone de tres datos,** correspondientes a la información de cada variable para cada observación.

Considera por ejemplo que, si falta un dato en alguna variable de alguna observación, se reportará como **NA**, mientras que si falta el nombre de una variable, **R** nos mostrará un **warning** para advertirnos y le asignará de forma automática un nombre.

```
read_csv("sexo, edad, ingreso
           1,40,
           2, 25, 2300
           1, 15, 4600")
```

```
## # A tibble:  3 x 3
##    sexo  edad  ingreso
##    <dbl>      <dbl>       <dbl>
## 1   1     40    NA
## 2   2     25    2300
## 3   1     15    4600
```

```
read_csv("sexo, edad,
           1,40,5000
           2, 25, 2300
           1, 15, 4600")
```

```
## Warning: Missing column names filled in: 'X3' [3]
## # A tibble: 3 x 3
```

```
##    sexo  edad  X3
##    <dbl> <dbl> <dbl>
## 1   1     40   5000
## 2   2     25   2300
## 3   1     15   4600
```

Dentro del funcionamiento de **read_csv**, podemos incluir algunas indicaciones extras. Por ejemplo; que omita determinadas líneas o que asigne determinados nombres a las columnas.

Consideremos que lo que necesitamos cargar contiene cierta información basura. Y que ya hemos identificado que esa información no deseada, comienza por ejemplo con **#**. Podemos usar la instrucción **comment="#"** para decirle que omita pasar esa información a **R**

```
read_csv("#Algo que no quiero en la base
     #Otro algo que tampoco quiero
     sexo, edad, ingreso
          1,40,5000
          2, 25, 2300
          1, 15, 4600
     #Otra cosa", comment="#")
```

```
## # A tibble: 4 x 3
##    sexo       edad       ingreso
##    <chr>      <chr>      <chr>
## 1  sexo       edad       ingreso
## 2  1          40         5000
## 3  2          25         2300
## 4  1          15         4600
```

Una opción alterna a **comment="#"** es usar **skip=n**, que indicará que se omitan las primeras n líneas. Por ejemplo

```
read_csv("#Algo que no quiero en la base
     #Otro algo que tampoco quiero
     sexo, edad, ingreso
     1,40,5000
     2, 25, 2300
     1, 15, 4600", skip=2)
```

```
## # A tibble: 3 x 3
##    sexo       edad       ingreso
##    <dbl>           <dbl>           <dbl>
## 1    1          40          5000
## 2    2          25          2300
## 3    1          15          4600
```

Un ejemplo más con el uso de ambas

```
read_csv("#Algo que no quiero en la base, basura 1, basura 2
     ?Otro algo que tampoco quiero, mas basura, basura1000
     NA, 50, NA
     sexo, edad, ingreso
     1,40,5000
     2, 25, 2300
     1, 15, 4600
     No te quiero en mis datos, a ti tampoco, a ti menos", skip=3, comment="N")
```

```
## # A tibble: 3 x 3

##    sexo  edad  ingreso
##    <dbl>      <dbl>      <dbl>
## 1    1   40    5000
## 2    2   25    2300
## 3    1   15    4600
```

Observa que si eliminamos las opciones **skip=3** y **comment="N"** del código anterior, tendremos una objeto que tendría la siguiente información.

```
## # A tibble: 7 x 3
##    `#Algo que no quiero en la base`     `basura 1`  `basura 2`
##    <chr>                             <chr>      <chr>
## 1  ?Otro algo que tampoco quiero        mas basura  basura1000
## 2  <NA>                               50         <NA>
## 3  sexo                              edad       ingreso
## 4  1                                 40         5000
## 5  2                                 25         2300
## 6  1                                 15         4600
## 7  No te quiero en mis datos         a ti tampo  a ti menos
```

Observa que en todos los ejemplos que hemos efectuado, el **contenido** de lo que deseamos este en la base, se encuentra separado por " ". Después hay una coma con el resto de las opciones.

En ocasiones puede darse el caso en el que la información no contenga el nombre las diferentes columnas o variables. Si ese es el caso, tendríamos dos opciones. La primera es decirle a **R** que no tenemos el nombre de las columnas en la base. Recuerda qué de manera automática, **R** entenderá siempre que la primera línea son los nombres. En este caso, se asigna de manera automática un nombre de columna como **X1, X2, X3**.

```
read_csv("1,40,5000
     2, 25, 2300
     1, 15, 4600", col_names=FALSE)
```

```
## # A tibble: 3 x 3
##      X1    X2    X3
##   <dbl> <dbl> <dbl>
## 1     1    40  5000
## 2     2    25  2300
## 3     1    15  4600
```

La segunda opción es asignar de manera manual, los nombres a las columnas.

```
read_csv("1,40,5000
     2, 25, 2300
     1, 15, 4600", col_names=c("sexo","edad","ingreso"))
```

```
## # A tibble: 3 x 3
##    sexo     edad ingreso
##   <dbl> <dbl> <dbl>
## 1   1    40    5000
## 2   2    25    2300
## 3   1    15    4600
```

Para asignar los nombres debemos usar la indicación **c("sexo","edad","ingreso")** en la cual cada nombre de cada columna se encuentra entre comillas y separados por una coma.

Al momento de estar escribiendo el código, no deseamos el contenido de cada observación en una línea, podemos usar el símbolo diagonal n, para indicar que se trata de un cambio de línea. Esto es

```
read_csv("1,40,5000\n2, 25, 2300\n1, 15, 4600", col_names=c("sexo","eda-
d","ingreso"))
```

```
## # A tibble: 3 x 3
##    sexo     edad ingreso
##   <dbl>       <dbl>  <dbl>
## 1   1    40    5000
## 2   2    25    2300
## 3   1    15    4600
```

Con estos elementos en mente, probemos ahora cargando una versión reducida de la base de datos **enoe**, utilizada en el capítulo 3. Tal y como lo hemos aplicado anteriormente, lo primero que debemos hacer es fijar el directorio de trabajo donde se encuentra la base que es de nuestro interés. Una vez que hemos efectuado este paso, lo siguiente es asignar el nombre de un objeto en **R** donde podamos almacenar la información contenida en la base de datos. Con fines prácticos usaremos la base **mu_enoe2** que previamente hemos trabajo, con la diferencia que ahora la información se encuentra en el archivo con extensión **.csv** y no se trata de la base completa.

```
setwd("~/Dropbox/Curso de R/Cap5_Importacion")
enoe <- read_csv("mu_enoe2.csv")
```

```
##
## -- Column specification --------------------------------------------------
-------------
## cols(
##   Mujer = col_character(),
##   `56` = col_double(),
## `  5375` = col_double()
## )
```

Observa que R nos indica que la base tiene tres columnas con nombres Mujer, 56 y 5375. Eso se debe a que la base no incluye los nombres de la columna, por lo tanto se importan de manera incorrecta. Para solucionarlo debemos hacer

```
enoe <- read_csv("mu_enoe2.csv",col_names=c("sexo","edad","ingreso") )
```

```
##
## -- Column specification --------------------------------------------------
-------------
## cols(
##   sexo = col_character(),
##   edad = col_double(),
##   ingreso = col_double()
## )
```

Una vez que el archivo ha sido leído, **R** nos mostrará una lista de las variables que contiene, así como el tipo de variable de que se trata. En este caso observamos que únicamente las variables **edad, ingreso** son de tipo **double o dbl**. Del capítulo anterior recordamos que eso significa que son **dbl: números reales**.

3 Análisis de un vector

Antes de continuar con los elementos para cargar bases de datos a R, conviene profundizar un poco en el uso de algunas funciones existentes dentro de readr; parse_logical, parse_integer, parse_date.

Cada una de estas tres funciones tiene como argumento principal, un vector que se desea analizar. Cabe señalar que el vector en cuestión contiene texto. Su función es tomar el vector que se proporciona en el argumento y convertirlo en un tipo especial de vector. De manera que parse_logical, lo transforma en un vector lógico, parse_integer en un vector de números enteros y parse_date en un vector de fechas.

Para ejemplificar este proceso, comencemos construyendo un vector

```
vector ←c ("TRUE","FALSE","NA")
class(vector)
```

```
## [1] "character"
```

```
class(parse_logical(vector))
```

```
## [1] "logical"
```

```
str(parse_logical(vector))
```

```
## logi [1:3] TRUE FALSE NA
```

Observa que construimos el objeto **vector** y antes de usar la función **parse_logical**, veíamos que era de tipo **character**, después de aplicarla se convierte en tipo **logical**. La función **str()** nos muestra la estructura interna de un objeto en **R**. En este caso nos indica que se trata de un vector lógico cuya dimensión consiste en una fila y tres columnas. **str()**, también nos muestra el contenido del vector.

Algo semejante sucede con las otras funciones. Observa los siguientes códigos:

```
vector2 ← c("1","2","3")
class(vector2)
```

```
## [1] "character"
```

```
class(parse_integer(vector2))
```

```
## [1] "integer"
```



```
str(parse_integer(vector2))
```

```
## int [1:3] 1 2 3
```

```
vector3 ← c("2010-01-01","1979-10-14")
class(vector3)
```

```
## [1] "character"
```

```
class(parse_date(vector3))
```

```
## [1] "Date"
```

```
str(parse_date(vector3))
```

```
## Date[1:2], format: "2010-01-01" "1979-10-14"
```

Cuando el vector de texto tiene elementos vacíos, podemos usar la opción na = ".", para indicarle a R que trate esos valores como nulos, los cuales usualmente se expresan como NA

```
parse_integer(c("1", "231", ".", "456"), na = ".")
```

```
## [1]     1    231   NA    456
```

Si tratamos de convertir en un entero, algo que no es un número o un entero expresado como texto, obtendremos una advertencia de parte de R, para indicarnos la inconsistencia. Considera por ejemplo el siguiente código

en el cual se nos indica que en los renglones 3 y 4 hay una inconsistencia.

```
error←parse_integer(c("123", "345", "abc", "123.45"))
```

```
## Warning: 2 parsing failures.
## row col         expected  actual
##    3 -- an integer         abc
##    4 -- no trailing characters  123.45
```

Este error ocasiona que los renglones 3 y 4 estén vacíos (es decir un **NA**)

Otro grupo de funciones con características similares, aunque con un grado de complejidad mayor a las anteriores son; **parse_double(), parse_number(), parse_character(), parse_factor(), parse_datetime()**. Cada una de estas funciones nos ayudará a analizar y modificar el contenido de lOs datos que importemos a **R**. Veamos más a detalle cada uno de ellos.

3.1 Números

Cuando trabajamos con bases de datos, quisiéramos que estuvieran adaptadas a nuestra necesidades y que pudiéramos pasar de inmediato al manejo y análisis de datos. Sin embargo, existe una variedad de situaciones que pueden llegar a complicar nuestro trabajo, sobre todo, si usamos bases que provienen de diferentes países o que contienen información que no ha sido manipulada para el análisis. Por ejemplo;

1. En México usamos la . para referirnos al punto decimal. En algunos otros lados se utiliza , (coma). Esto de entrada puede significar un problema en el manejo de datos
2. Imagina una base de datos que tiene el ingreso mensual de las personas, pero incluye el símbolo $ o que tiene porcentajes e incluye %
3. En ocasiones se utilizan caracteres para abreviar un dato. Puede ser muy complicado escribir 1,000,000 y entonces se usan abreviaciones 1e+06.

Para solucionar estos problemas podemos usar lo siguiente;

```
x<-"1.23"
class(x)
```

```
## [1] "character"
```

```
y<-parse_double(x)
class(y)
```

```
## [1] "numeric"
```

```
# Si tenemos , en lugar de punto
x<-"1,23"
class(x)
```

```
## [1] "character"
```

```
y<-parse_double(x, locale = locale(decimal_mark = ","))
class(y)
```

```
## [1] "numeric"
```

```
y
```

```
## [1] 1.23
```

En este caso hemos agregado la indicación locale = locale(decimal_mark = ",") para informar a **R** que el dato tiene una , (coma) la cual debe ser considerada como punto .

Para el segundo caso

```
x<-"$100"
class(x)
```

```
## [1] "character"
```

```
y<-parse_number(x)
class(y)
```

```
## [1] "numeric"
```

```
x<-"30%"
class(x)
```

```
## [1] "character"
```

```
y<-parse_number(x)
class(y)
```

```
## [1] "numeric"
```

```
y
```

```
## [1] 30
```

```
x<-"La utilidad del mes fue de $15000"
class(x)
```

```
## [1] "character"
```

```
y<-parse_number(x)
class(y)
```

```
## [1] "numeric"
```

```
y
```

```
## [1] 15000
```

Observa que en todos los casos el objeto x contiene caracteres no validos para el análisis con datos, y la indicación **parse_number(x)** los transforma en un número que podemos usar.

Supón ahora que nuestra base de datos tiene números de las siguiente forma **$442.185.895.145** donde se ha usado . en lugar de , .En este caso al usar la función **parse_number(x)** obtendremos

```r
x<-"$442.185.895.145"
y<-parse_number(x)
y
```

```
## [1] 442.185
```

Esto representa un problema, pues sabemos que ese no es el número correcto. En este caso, debemos indicar que existe un marca de agrupación. Es decir

```r
x<-"$442.185.895.145"
y<-parse_number(x, locale = locale(grouping_mark = "."))
y
```

```
## [1] 442185895145
```

Observa que hemos usado la configuración con la función locale. Esto significa que estas configuraciones son específicas a una configuración **local** del sistema.

3.2 Cadenas de texto

El principal problema al leer cadena de texto radica en el **encoding** sobre el cual se encuentran los datos.

Por ejemplo, en inglés no existe la **ñ**. Si **R** tiene la configuración adecuada para leer esos caracteres, no tendrás problema alguno. Sin embargo, sí la configuración no es adecuada, verás algo como esto:

```r
x <- "El Ni\xf1o ha sido muy malo este a\xf1o"
```

Para corregir ese problema debemos indicar cual es el **encoding** sobre el cual se encuentran los datos. En el ejemplo anterior que acabamos de proporcionar los datos se encuentran bajo en **encoding ISO-8859-1**.

Sabemos esto, gracias a la función

```r
guess_encoding(charToRaw(x))
```

```
## # A tibble: 2 x 2
##   encoding   confidence
##   <chr>      <dbl>
## 1 ISO-8859-1 0.61
## 2 ISO-8859-2 0.35
```

La cual nos indica los probables **encodings** en los que se encuentran los datos. Cuando identificamos el **encoding** podemos corregir el problema usando la instrucción:


```r
parse_character(x, locale = locale(encoding = "ISO-8859-1"))
```

```
## [1] "El Niño ha sido muy malo este año"
```

En este ejemplo, sí cuando escribiste **x <- "El Ni\xf1o ha sido muy malo este a\xf1o** no tuviste ningún problema y de inmediato **R** reconoció **\xf1** como **ñ**, significa que **R** previamente ha sido configurado.

Uno de los problemas más comunes se relaciona con la inclusión de acentos en el texto. En ocasiones es mejor eliminarlos para evitar problemas. Por ejemplo;

```r
z<-"Inglés, Sarampión, Niño"
y<-chartr('áéíóúñ','aeioun', z)
y
```

```
## [1] "Ingles, Sarampion, Nino"
```

Habrá ocasiones donde necesitamos conservar los acentos, porque son importantes para un reporte. En esos casos, desde la consola debemos ejecutar una de las siguientes instrucciones, dependiendo el sistema operativo con el cual estemos trabajando.

- Para Windows Sys.setlocale("LC_ALL", "ES_ES.UTF-8")
- Para Mac options(encoding = 'UTF-8')

Los problemas de **encoding** son muy comunes al trabajar con bases de datos que contienen cadenas de texto. Actualmente se trabaja con un **encoding** que está mas o menos estandarizado; **UTF-8**. La librería **readr** usa este **enconding**, y asume que tus datos se encuentran bajo este formato. Si en algún momento al leer tus datos, te das cuenta que lucen extraños, seguramente tienes un problema de **encoding**. Resolverlos no es una tarea sencilla, y cada problema es específico a la base de datos con la que se está trabajando. Te recomendamos que revises eel siguiente documento, donde encontrarás algunas recomendaciones cuando te enfrentes a problemas de este tipo. Encoding

3.3 Factores

Dentro de **readr** existe una función que nos permite identificar si una palabra determinada corresponde a una lista nombres de una variable categórica. Por ejemplo, supón que tienes un vector de animales que contiene la siguiente información

```r
animales<-c("perro", "gato", "rata", "liebre", "hormiga")
```

Luego imagina que alguien te proporciona una lista de palabras, las cuales es de tu interés verificar si corresponden a la categoría de animales. La lista proporcionada es esta

```r
lista<-c("perro", "gato", "rrat", "libre", "perro")
```

Observa que dentro de la lista hay dos palabras que no corresponde con las que están establecidas en la categoría. Verificar esto es una tarea sencilla cuando se trata de vectores con pocas palabras, sin embargo, al contar con grandes cantidades de información, la tarea puede ser muy complicada. Para ello usamos **parse_factor();**

```
parse_factor(lista, levels = animales)
```

```
## Warning: 2 parsing failures.
##   row col           expected actual
##    3 -- value in level set  rrat
##    4 -- value in level set  libre
## [1] perro gato  <NA>  <NA>  perro
## attr(,"problems")
## # A tibble: 2 x 4
##    row   col expected          actual
##    <int> <int> <chr>               <chr>
## 1   3    NA    value in level set    rrat
## 2   4    NA    value in level set    libre
##  Levels:      perro gato rata liebre      hormiga
```

En este caso hemos solicitado que compare los elementos de la **lista**, con lo que contiene **animales**. La respuesta de **R** es que existe una inconsistencia, en las filas 3 y 4, ya que contienen palabras que no están presentes en la categoría de animales.

Una vez que identificamos que hay errores, podemos corregirlos haciendo uso de **gsub()**

```
lista←gsub("rrat", "rata", lista)
lista←gsub("libre", "liebre", lista)
```

Una vez efectuada la corrección, al correr nuevamente la validación obtenemos;

```
parse_factor(lista, levels = animales)
```

```
## [1] perro gato rata liebre perro
## Levels: perro gato rata liebre hormiga
```

3.4 Fechas y tiempos

Existen tres opciones para transformar caracteres a fechas, dependiendo si deseamos o no incluir el tiempo.


```
x←"20201001T2010"
class(x)
```

```
## [1] "character"
```

```
y←parse_datetime(x)
class(y)
```

```
## [1] "POSIXct" "POSIXt"
```

```
y
```

```
## [1] "2020-10-01 20:10:00 UTC"
```

Con esta indicación hemos transformado un vector de texto, a fecha. Observa que **x<-"2020-10-01T2010"** conserva el formato año (4 dígitos), mes (dos dígitos), día (dos dígitos), seguido de **T** (referente al tiempo) el cual esta en formato hora (dos dígitos), minuto (dos dígitos). Esta información puede estar separada por un -. / o sin ninguna separación.

En caso de que el vector de texto contenga sólo fechas y no tiempo, tenemos

```
x←"20201001"
class(x)
```

```
## [1] "character"
```

```
y←parse_datetime(x)
class(y)
```

```
## [1] "POSIXct" "POSIXt"
```

```
y
```

```
## [1] "2020-10-01 UTC"
```

En este caso de manera automática, se ha asignado una hora a la fecha, la cual corresponde a la media noche. Si deseamos obtener sólo la fecha y no la hora de un vector de texto usamos, **parse_date()**

```
x←"2020/10/01"
class(x)
```

```
## [1] "character"
```

```
y←parse_date(x)
class(y)
```

```
## [1] "Date"
```

```
y
```

```
## [1] "2020-10-01"
```

En ocasiones las lecturas de las fechas dependen de la región donde se escriban. Por ejemplo, en algunos países primero se escribe el mes, después el día y al final el año. **parse_date()** permite hacer estas adaptaciones. Considera el siguiente caso en el que especificamos la fecha en texto de forma incorrecta;

```
x←"09/12/20"
parse_date(x)
```

```
## Warning: 1 parsing failure.
##  row col      expected   actual
##    1   --      date like 09/12/20
## [1] NA
```

La instrucción nos arroja un error, pues el formato de año debe ser en cuatro dígitos. Podemos resolver esto simplemente indicando que el formato de nuestra fecha es diferente, haciendo:

```
x←"09/12/20"
parse_date(x, "%m/%d/%y")
```

```
## [1] "2020-09-12"
```

```
parse_date(x, "%d/%m/%y")
```

```
## [1] "2020-12-09"
```

```
parse_date(x, "%y/%m/%d")
```

[1] "2009-12-20"

En el primer caso hemos indicado que considere la primera entrada como un mes, la segunda como un día y la última como el año. En el segundo caso hemos cambiado el ordenamiento. Observa qué en todos los casos, la función nos regresa en el formato año, mes y día.

Esto es posible gracias a que la función reconoce los siguientes patrones:

- %Y identifica que el año está en formato de 4 dígitos
- %y identifica que el año está en formato de 2 dígitos

A diferencia **parse_datetime(), parse_date()** exige una separación entre cada rubro de fecha. Prueba haciendo lo siguiente para que lo compruebes;

```
x←"20201001"
parse_datetime(x)
```

```
## [1] "2020-10-01 UTC"
```

```
parse_date(x)
```

```
## Warning: 1 parsing  failure.
##  row col      expected   actual
##    1   --     date like  20201001
## [1] NA
```

Finalmente tenemos **parse_time()** el cual transformará una cadena de texto en un formato de tiempo. Este función espera un formato de texto, dado por; **horas:minutos: segundos**. De manera opcional se puede especificar am/pm.

```
x←"05:30:05 pm"
class(x)
```

```
## [1] "character"
```

```
y←parse_time(x)
class(y)
```

```
## [1] "hms"     "difftime"
```

```
y
```

```
## 17:30:05
```

4 Análisis de un archivo

Con los elementos que hemos aprendido sobre la lectura y transformación de datos, consideremos nuevamente el problema de cargar un archivo de datos a **R**, con el uso de la función **readr.** Cuando cargamos un archivo usando las funciones de esta librería se analizan las primeras 1000 filas para tratar de determinar el tipo de dato de que se trata. Cada dato toma una de las siguientes clasificaciones, dependiendo del contenido de las observaciones que analiza;

- logical: Si contienen únicamente “F”, “T” o “FALSO”, “VERDADERO”

- integer: Si contienen únicamente caractéres numéricos y -
- double: Si contienen únicamente números reales
- time: Se ajusta a un formato de tiempo
- date: Se ajusta a un formato de fecha
- date-time: Se ajusta a un formato de tiempo y fecha
- character: Cuando no se ajusta a ninguno de los formatos anteriores.

Limitar la clasificación de cada variable con el análisis únicamente de los primeros 1000 datos puede resultar un tanto inadecuado.

Para analizar esto consideremos una base de datos de ejemplo que tiene precisamente **readr** para ayudarnos a entender el proceso de importación.

Ejecuta el siguiente código

```
prueba <- read_csv(readr_example("challenge.csv"))
##
## -- Column specification ------------------------------------------------
-------------
## cols(
##   x = col_double(),
##   y = col_logical()
## )
## Warning: 1000 parsing failures.
## row     col        expected    actual
## 1001     y  1/0/T/F/TRUE/FALSE  2015-01-16  '/Library/Frameworks/R.framework/
Versions/4.0/Resources/library/
## 1002     y  1/0/T/F/TRUE/FALSE  2018-05-18  '/Library/Frameworks/R.framework/
Versions/4.0/Resources/library/
## 1003     y  1/0/T/F/TRUE/FALSE  2015-09-05  '/Library/Frameworks/R.framework/
Versions/4.0/Resources/library/
## 1004     y  1/0/T/F/TRUE/FALSE  2012-11-28  '/Library/Frameworks/R.framework/
Versions/4.0/Resources/library/
## 1005     y  1/0/T/F/TRUE/FALSE  2020-01-13  '/Library/Frameworks/R.framework/
Versions/4.0/Resources/library/
## .... ... .................. .......... ..................................
...................................................
## See problems(...) for more details.
```


readr_example("challenge.csv") únicamente regresa una trayectoria de donde se encuentra el archivo **challenge,** el cual se encuentra dentro del contenido de la librería **readr**.

Observa que el resultado nos indica que la primera columna es de tipo **double**, mientras que la segunda es de tipo **logical**. Observa también que hay una indicación que nos advierte que en la **row** 1001 se esperaba **1/0/T/F/TRUE/FALSE** (debido a que con las primeras 1000 observaciones dedujo que la variable era de tipo logical) y que la información contiene **2015-01-16.** La misma indicación se repite para varias filas.

Para profundizar un poco en el problema, usemos la función **problems()**

```
problems(prueba)
## # A tibble: 1,000 x 5
##     row  col   expected          actual      file
##    <int> <chr> <chr>               <chr>       <chr>
## 1  1001  y     1/0/T/F/TRUE/~ 2015-01~ '/Library/Frameworks/R.framework/Ver-
sion~
## 2  1002  y     1/0/T/F/TRUE/~ 2018-05~ '/Library/Frameworks/R.framework/Ver-
sion~
## 3  1003  y     1/0/T/F/TRUE/~ 2015-09~ '/Library/Frameworks/R.framework/Ver-
sion~
## 4  1004  y     1/0/T/F/TRUE/~ 2012-11~ '/Library/Frameworks/R.framework/Ver-
sion~
## 5  1005  y     1/0/T/F/TRUE/~ 2020-01~ '/Library/Frameworks/R.framework/Ver-
sion~
## 6  1006  y     1/0/T/F/TRUE/~ 2016-04~ '/Library/Frameworks/R.framework/Ver-
sion~
## 7  1007  y     1/0/T/F/TRUE/~ 2011-05~ '/Library/Frameworks/R.framework/Ver-
sion~
## 8  1008  y     1/0/T/F/TRUE/~ 2020-07~ '/Library/Frameworks/R.framework/Ver-
sion~
## 9  1009  y     1/0/T/F/TRUE/~ 2011-04~ '/Library/Frameworks/R.framework/Ver-
sion~
## 10 1010  y     1/0/T/F/TRUE/~ 2010-05~ '/Library/Frameworks/R.framework/Ver-
sion~
## # ... with 990 more rows
```

Esto nos muestra a detalle donde se encuentran los problemas, indicando la fila y la columna. Este análisis nos indica que el conflicto se encuentra en la columna (variable) de nombre *y*. Podemos intentar importar nuevamente la base de datos considerando ahora que quizá la columna de nombre y es más bien de tipo **fecha** y no **logical.**

```
prueba <- read_csv(readr_example("challenge.csv"),
           col_types = cols(
                 x=col_double(),
                 y=col_date()
                 ))
```

En esta ocasiones cuando cargamos los datos, **R** no tuvimos ningún tipo de advertencia. Para asegurarnos que se cargó correctamente analizamos las primeras y las últimas observaciones de la base. Esto lo podemos hacer gracias a **head y tail**

```
head(prueba)
```

```
## # A tibble: 6 x 2
##       x     y
##    <dbl> <date>
## 1  404   NA
## 2  4172  NA
## 3  3004  NA
## 4  787   NA
## 5   37   NA
## 6  2332  NA
```

```
tail(prueba)
```

```
## # A tibble: 6 x 2
##    x      y
##    <dbl> <date>
## 1  0.805  2019-11-21
## 2  0.164  2018-03-29
## 3  0.472  2014-08-04
## 4  0.718  2015-08-16
## 5  0.270  2020-02-04
## 6  0.608  2019-01-06
```

Observa que en los primeros valores de la base (head) obtenemos **NA** para el caso de la columna **y**. Esto es porque efectivamente no hay datos en esas observaciones. Sí ejecutamos ahora la inspección de problemas, tenemos;


```
problems(prueba)
```

```
## [1] row col   expected actual
## <0 rows> (or 0-length row.names)
```

Indicando que no tenemos ningún problema en la importación.

La recomendación es que siempre que se importe una base de datos, especificar el tipo de información de que se trata, es decir, usar siempre col_types. Cada uno de los diferentes **parse_** que hemos aprendido, tiene su correspondiente **col_**, para facilitar la importación.

Si no tenemos información sobre los tipos de variables que vamos a importar podemos usar **cols(.default = col_character())** para que todas las columnas se importen como texto y nos permitan obtener mas

información sobre ellas. En este caso;

```
prueba2 <- read_csv(readr_example("challenge.csv"),
col_types = cols(.default = col_character())
)
```

En el ejemplo que hemos usado exactamente el error se ubicaba en la fila 1,001 y como **readr** únicamente consideraba las primeras 1,000, no se efectuó una correcta identificación. Si deseamos que **readr** analice mas de 1,000 observaciones podemos indicarlo usando guess_max. En el ejemplo siguiente cargaremos nuevamente la base, pero ahora le diremos que analice hasta la fila 1,001

```
prueba3 <- read_csv(readr_example("challenge.csv"), guess_max = 1001
)
```

```
##
## -- Column specification ---------------------------------------------------
-------------
## cols(
##    x = col_double(),
##    y = col_date(format = "")
## )
```

```
problems(prueba3)
```

```
## [1] row col   expected actual
## <0 rows> (or 0-length row.names)
```

Observa qué en este caso, no tenemos ningún error y de manera automática, la base se ha cargado correctamente, debido a que se ha considerado hasta el dato 1001, para determinar el tipo de variable.

La base que hemos usado contiene únicamente dos columnas y solo una de ellas presentó problemas en el momento de la importación. En caso de haber más columnas defectuosas, se recomienda corregir una a la vez, con el procedimiento descrito.

En ocasiones la base de datos sobre la cual deseamos trabajar puede ser sumamente grande. En esos casos conviene indicar únicamente un conjunto de algunas pocas observaciones para el análisis antes de cargar toda la base completa. Para ello podemos usar **n_max = n**, indicación con la cual se cargarán solamente las primeras n observaciones. Esto es;

```
prueba4 <- read_csv(readr_example("challenge.csv"), guess_max = 1001, n_max =
200)
```

```
##
## -- Column specification --------------------------------------------------
-------------
## cols(
##   x = col_double(),
##   y = col_date(format = "")
## )
```

Otras ocasiones conviene únicamente leer en líneas para identificar el tipo de información en cada variable. Esto permite agrupar en una sola línea toda la información correspondiente a cada observación separada con comas. Para ello usamos **read_lines()**

```
w<- read_lines(readr_example("challenge.csv"))
w[1007]
```

```
## [1] "0.473980569280684,2016-04-17"
```

Observa que hemos leído en líneas y al solicitar que nos muestra la línea 1007 con la indicación w[1007]. Esto nos permite darnos cuenta que tenemos un vector de texto, al cual podemos aplicar los procedimientos que hemos aprendido en este apartado.

5 Exportar datos

En ocasiones será necesario enviar de regreso una base de datos, después de que ha sido manipulada aformatos que se puedan leer en otros programas. En **readr**, existen dos funciones para ese fin.

La primera de ella es **write_csv().** Lamentablemente cuando exportamos los datos usando esta función, se pierde el formato que previamente habíamos identificado de las variables. En consecuencia, en un futuro cuando deseemos cargarlos nuevamente tendremos que volver efectuar la corrección de tipo de variable que hicimos. Observa esto con el siguiente código, donde primero

exportamos los datos usando **write_csv(),** seguido con el nombre del objeto en **R** y el nombre que deseamos darle entre comillas. Después lo importamos nuevamente y observamos el mismo problema que anteriormente.

```
write_csv(prueba3, "challenge.csv")
read_csv("challenge.csv")
```

```
##
## -- Column specification --------------------------------------------------
-------------
## cols(
##   x = col_double(),
##   y = col_logical()
## )
## Warning: 1000 parsing failures.
## row     col        expected          actual          file
## 1001     y     1/0/T/F/TRUE/FALSE 2015-01-16 'challenge.csv'
## 1002     y     1/0/T/F/TRUE/FALSE 2018-05-18 'challenge.csv'
## 1003     y     1/0/T/F/TRUE/FALSE 2015-09-05 'challenge.csv'
## 1004     y     1/0/T/F/TRUE/FALSE 2012-11-28 'challenge.csv'
## 1005     y     1/0/T/F/TRUE/FALSE 2020-01-13 'challenge.csv'
## .... ... .................. .......... ...............
## See problems(...) for more details.
## # A tibble: 2,000 x 2
##   x     y
##   <dbl> <lgl>
## 1  404   NA
## 2  4172  NA
## 3  3004  NA
## 4  787   NA
## 5  37    NA
## 6  2332  NA
## 7  2489  NA
## 8  1449  NA
## 9  3665  NA
## 10 3863  NA
## # ... with 1,990 more rows
```

La segunda función presente en **readr** para exportar datos es **write_tsv()**. Observa que después de exportar y volver importar, sucede lo mismo que con la función **write_csv()**

```
write_tsv(prueba3, "challenge.tsv")
read_tsv("challenge.tsv")
```

```
##
## -- Column specification ------------------------------------------------
-------------
## cols(
##   x = col_double(),
##   y = col_logical()
## )
## Warning: 1000 parsing failures.
## row     col              expected    actual           file
## 1001    y     1/0/T/F/TRUE/FALSE 2015-01-16 'challenge.tsv'
## 1002    y     1/0/T/F/TRUE/FALSE 2018-05-18 'challenge.tsv'
## 1003    y     1/0/T/F/TRUE/FALSE 2015-09-05 'challenge.tsv'
## 1004    y     1/0/T/F/TRUE/FALSE 2012-11-28 'challenge.tsv'
## 1005    y     1/0/T/F/TRUE/FALSE 2020-01-13 'challenge.tsv'
## .... ... .................. .......... ...............
## See problems(...) for more details.
## # A tibble: 2,000 x 2
##      x    y
##    <dbl> <lgl>
## 1  404   NA
## 2  4172  NA
## 3  3004  NA
## 4  787   NA
## 5  37    NA
## 6  2332  NA
## 7  2489  NA
## 8  1449  NA
## 9  3665  NA
## 10 3863  NA
## # ... with 1,990 more rows
```


Para que esto no suceda y podamos conservar la corrección efectuada sobre el tipo de variables, debemos hacer lo siguiente;

```
write_rds(prueba3, "challenge.rds")
read_rds("challenge.rds")
```

```
## # A tibble: 2,000 x 2
##     x     y
##    <dbl>      <date>
## 1  404   NA
## 2  4172  NA
## 3  3004  NA
## 4  787   NA
## 5  37    NA
## 6  2332  NA
## 7  2489  NA
## 8  1449  NA
## 9  3665  NA
## 10 3863  NA
## # ... with 1,990 more rows
```

Observa que hemos dado salida a los datos en un formato **rds** que almacena las característica específicas del tipo de variable. Obviamente la lectura de vuelta de este archivo debe hacerse con **read_rds**, ya que el archivo exportado, ya no es un csv, pues ahora es rds. La extensión **rds** es un tipo especial de formato en **R**.

En los casos anteriores el archivo exportado tiene una extensión que definirá su uso en otros lenguajes y programas. Existe un formato que es compatible entre lenguajes de programación. Nos referimos a **feather**. Para ello es necesario instalar la librería y activarla para su uso en la sesión de trabajo.

```
library(feather)
write_feather(prueba3, "challenge.feather")
read_feather("challenge.feather")
```

```
## # A tibble: 2,000 x 2
##    x     y
##    <dbl> <date>
## 1  404   NA
## 2  4172  NA
```

```
## 3  3004  NA
## 4  787   NA
## 5  37    NA
## 6  2332  NA
## 7  2489  NA
## 8  1449  NA
## 9  3665  NA
## 10 3863  NA
## # ... with 1,990 more rows
```

Observa que en este caso no hemos indicado una extensión del archivo. Los archivos con este formato tienen un mayor grado de compatibilidad con otros programas.

6 Importar otros tipos de datos

R nos permite cargar otros tipos de datos. Por ejemplo, datos que provienen de programas como SPSS, STATA o SAS puede cargarse usando la librería **haven**. En el caso de los formatos de EXCEL, previamente ya hemos cargado datos con **readxl**.

A través de los siguientes capítulos usaremos bases de datos que se encuentran en distintos formatos, lo que nos permitirá poner a prueba los conocimientos adquiridos.

7 Tibbles y data frame

A lo largo de este capítulo y en algunos puntos de los capítulos pasados, hemos observado que después de leer un conjunto de datos, ya sea desde un archivo o desde un vector creado por nosotros, **R** nos muestra una indicación como esta **## # A tibble: 3 x 3**. Obsérvala en la primera línea después de ejecutar este código.

```
read_csv("sexo, edad, ingreso
     1,40,5000
     2, 25, 2300
     1, 15, 4600")
```

```
## # A tibble: 3 x 3
##    sexo  edad  ingreso
##    <dbl>       <dbl>       <dbl>
## 1   1    40    5000
## 2   2    25    2300
## 3   1    15    4600
```


Lo que nos indica es que se trata de un objeto tipo **tibble** que tiene tres filas y tres columnas. Un **tibble** esun tipo de **data frame** mas moderno en el que se modifican algunos atributos para hacer el trabajo mas fácil. En este capítulo nos enfocaremos en señalar las diferencias entre los **data frame** y los **tibble**. Ambos tipos de objetos son ampliamente usados cuando se trabaja con **R**.

Consideremos crear una pequeña base de datos, que contenga información. Para ello vimos que podemos usar read_csv como en el ejemplo anterior. El mismo ejemplo lo podemos replicar usando **tibble** y **data_frame**

```
tibble(
     sexo=c(1,2,1),
     edad=c(40,25,15),
     ingreso=c(5000,2300,4600)
          )
```

```
## # A tibble: 3 x 3
##    sexo  edad  ingreso
##    <dbl>       <dbl>       <dbl>
## 1   1    40    5000
## 2  2     25    2300
## 3  1     15    4600
```

```
data_frame(
     sexo=c(1,2,1),
     edad=c(40,25,15),
     ingreso=c(5000,2300,4600)
          )
```

```
## Warning: `data_frame()` is deprecated as of tibble 1.1.0.
## Please use `tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
## # A tibble: 3 x 3
##    sexo  edad  ingreso
##    <dbl>       <dbl>       <dbl>
## 1   1    40    5000
## 2   2    25    2300
## 3   1    15    4600
```

Hasta ahora parece que no hay diferencias en el uso de **tibble** o **data_frame** y solo obtenemos un *warnings* en el caso del **data_fram** indicando que esta función es obseleta y que usemos **tibble** en su lugar. Aún así se efectua la instrucción y se genera la pequeña base.

Probemos ahora con los siguiente tres ejemplos usando **tibble**.

```
tibble(
x = 1:5,
y = 1,
z = x ^ 2 + y
)
```

```
## # A tibble: 5 x 3
##    x     y     z
##    <int> <dbl> <dbl>
## 1  1     1     2
## 2  2     1     5
## 3  3     1     10
## 4  4     1     17
## 5  5     1     26
```

```
tibble(
`:)` = "smile",
` ` = "space",
`2000` = "number"
)
```

```
## # A tibble: 1 x 3
##    `:)` ` ` `2000`
##    <chr> <chr> <chr>
## 1  smile space number
```

```
tribble(
~x, ~y, ~z,
#--|--|----
"a", 2, 3.6,
"b", 1, 8.5
)
```

```
## # A tibble: 2 x 3
##     x    y           z
##    <chr> <dbl>      <dbl>
## 1   a    2          3.6
## 2   b    1          8.5
```


Si tratas de replicar estos ejemplos cambiando **tibble** por **data_frame**, te darás cuenta qué en el último ejemplo, el data frame no funciona. Esto nos ayuda a entender que **tibble** es mucho mas flexible con la construcción de estructuras de datos. Observa que fue posible usar caracteres especiales para nombrar las columnas gracias al uso de apostrofes.

La diferencia mas importante entre el uso de estas dos funciones se observa cuando pedimos ver el contenido de un **data frame** o **tibble**.

7.1 Imprimir en Pantalla

Para ver estas diferencias, consideremos una base de datos, propia de **R**, para el aprendizaje. Esta base de datos se llama **flights** y se encuentra en la librería de nombre **nycflights13**. Podemos acceder a ella haciendo **nycflights13::flights**. Usaremos también la base de datos **iris**.

Para empezar, veamos de que clase de objeto se trata cada una de ellas.

```
class(nycflights13::flights)
```

```
## [1] "tbl_df"  "tbl"       "data.frame"
```

```
class(iris)
```

```
## [1] "data.frame"
```

La primera tiene un formato **tibble**, mientrasque la segunda es de tipo **data frame**.

Observa que en este caso debido a que la base de datos **flights** es **tibble**, me muestra sólo algunos de los elementos de la base.

```
nycflights13::flights
```

```
## # A tibble: 336,776 x 19
## year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int>            <int>      <int>      <int>      <int>      <dbl>
<int>      <int>
## 1  2013        1      1     517    515    2      830   819
## 2  2013        1      1     533    529    4      850   830
## 3  2013        1      1     542    540    2      923   850
## 4  2013        1      1     544    545    -1     1004  1022
## 5  2013        1      1     554    600    -6     812   837
## 6  2013        1      1     554    558    -4     740   728
## 7  2013        1      1     555    600    -5     913   854
## 8  2013        1      1     557    600    -3     709   723
```

```
## 9  2013       1     1     557   600   -3    838   846
## 10 2013       1     1     558   600   -2    753   745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## # carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## # air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

En el caso de la base de datos **iris** que es **data frame**, me muestra todos los elementos de la base.

```
iris
```

En este caso hemos omitido los resultados, pero puedes probarlo en tu consola y te darás cuenta de que enlista todos los elementos.

Esa es una diferencia fundamental, entre ambos tipos de objetos. Aunque parece sencilla, toma relevancia cuando se trabajan con bases de datos que contiene grandes cantidades de información.

Podemos convertir un data frame en tibble, para ello hacemos;

```
iris2<-as_tibble(iris)
iris2
```

```
## # A tibble: 150 x 5
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##    <dbl>       <dbl>       <dbl>       <dbl>       <fct>
## 1  5.1   3.5   1.4   0.2   setosa
## 2  4.9   3     1.4   0.2   setosa
## 3  4.7   3.2   1.3   0.2   setosa
## 4  4.6   3.1   1.5   0.2   setosa
## 5  5 3.  6 1.  4     0.2   setosa
## 6  5.4   3.9   1.7   0.4   setosa
## 7  4.6   3.4   1.4   0.3   setosa
## 8  5 3.  4 1.  5     0.2   setosa
## 9  4.4   2.9   1.4   0.2   setosa
## 10 4.9   3.1   1.5   0.1   setosa
## # ... with 140 more rows
```

En ocasiones, aunque tengamos un formato tipo **tibble** nos gustaría observar solo un pequeño conjunto de observaciones. En esos casos podemos hacer:


```
nycflights13::flights %>%
print(n = 5, width = Inf)
```

```
## # A tibble: 336,776 x 19
## year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int>             <int>      <int>      <int>             <int>
<dbl>             <int>      <int>
## 1  2013       1     1     517         515          2          830   819
## 2  2013       1     1     533         529          4          850   830
## 3  2013       1     1     542         540          2          923   850
## 4  2013       1     1     544         545         -1          1004  1022
## 5  2013       1     1     554         600         -6          812   837
## arr_delay carrier flight tailnum origin dest air_time distance hour minute
##  <dbl> <chr>  <int>       <chr>      <chr> <chr> <dbl> <dbl>
<dbl>       <dbl>
## 1  11 UA      1545  N14228     EWR   IAH   227   1400       5      15
## 2  20  UA     1714  N24211     LGA   IAH   227   1416       5     29
## 3  33  AA     1141  N619AA     JFK   MIA   160   1089       5     40
## 4  -18 B6     725   N804JB     JFK   BQN   183   1576       5     45
## 5  -25 DL     461   N668DN     LGA   ATL   116   762        6     0
## time_hour
## <dttm>
## 1 2013-01-01 05:00:00
## 2 2013-01-01 05:00:00
## 3 2013-01-01 05:00:00
## 4 2013-01-01 05:00:00
## 5 2013-01-01 06:00:00
## # ... with 336,771 more rows
```

Esto nos mostrará únicamente las primeras cinco observaciones y todas las columnas. Observa que hemos usado el operador **pipe** utilizado anteriormente. Esta indicación funciona también con la base de **iris**.

7.2 Hacer subconjuntos

En ocasiones vamos a requerir trabajar específicamente con todas las observaciones de una variable. Cuando trabajamos con un objeto del tipo **tibble** o **data frame** podemos hacer lo siguiente:

```
# Partimos del objeto
df<-tibble(
x = 1:5,
y = 1,
z = x ^ 2 + y
)
df
```

```
## # A tibble: 5 x 3
##     x     y     z
##   <int> <dbl>      <dbl>
## 1  1     1     2
## 2  2     1     5
## 3  3     1     10
## 4  4     1     17
## 5  5     1     26
```

```
df$z
```

```
## [1] 2 5 10 17 26
```

Observa que al indicar **df$z** hemos seleccionado únicamente la columna **z** de todo el objeto **df**. Una forma semejante de efectuar esta selección es usar **df[[3]]**. En este caso usamos 3, debido a que la columna **z** es la tercera. Lo mismo sucederá usando **df[["z"]].**

```
df[[3]]
```

```
## [1]      2     5      10     17     26
```

```
df[["z"]]
```

```
## [1]      2     5      10     17     26
```

Si deseamos usar la notación con el operador pipe, tendríamos

```
df %>% .$z
```



```
## [1]      2     5      10     17     26
```

```
df %>% .[[3]]
```

```
## [1]      2     5      10     17     26
```

```
df %>% .[["z"]]
```

```
## [1]      2     5      10     17     26
```

8 Actividades

1. Identifica el error en la lectura de cada uno de los siguientes vectores. Propón una solución para que puedan cargarse correctamente

• read_csv("a,b\n1,2,3\n4,5,6")

• read_csv("a,b,c\n1,2\n1,2,3,4")

• read_csv("a,b\n\"1")

• read_csv("a,b\n1,2\na,b")

• read_csv("a;b\n1;3")

2. ¿Qué sucede si fijamos la configuración decimal_mark() y grouping_mark() en el mismo vector?

3. Construye el siguiente data frame. Usa tibble. Después crea un objeto que contenga únicamente la variable total.

educación	hogar	total
primaria	familiar	5000
secundaria	jefe mujer	2300
primaria	jefe mujer	4600

4. Dentro de las bases de datos de ejemplo que están disponibles en readr se encuentra la base de nombre **massey-rating.txt**.

• Usa la función **read_csv** para importar el archivo. ¿Qué errores detectan en la importación? Usa el comando **View()** para visualizar completamente el contenido de la base.

• Ademas del formato csv, en ocasiones es común encontrar bases de datos en formato txt. Para leer este tipo de archivos usamos la indicación **read_table**. Usa esta función para importar el archivo. ¿Mejoró la importación?

• Indica cuales elementos de la lista no pertenecen a los diferentes equipos contenidos en la variable **Team** de la base **massey-rating.txt**

```
lista<-c("TCU","Georgia Tech","Mississippi","Baylor","Michigan St","Ohio")
```

Capítulo 6

Datos ordenados

Capítulo 6 | Datos ordenados

1 Introducción

En este capítulo trabajaremos con datos ordenados, o tablas de datos. Los datos ordenados, generalmente, se refieren a una tabla de datos que tiene cierto formato para analizar la información. Es decir, filas y columnas que especifican la información de individuos (u objetos) y variables. Una misma tabla de datos puede presentarse de diversas formas, pero no todas las formas o estructuras de tablas son eficientes para generar visualizaciones o generar agregados estadísticos. En general, será necesario manipular los datos para generar una tabla estructurada u organizada a la que llamaremos *tidy data*.

El término *tidy data* se refiere a una forma específica de organizar los valores de un conjunto de datos. Una tabla de datos del tipo *tidy data* se organiza de la siguiente manera: las filas representan individuos u observaciones, mientras que las columnas representan variables y los valores referentes a cada observación en cada variable se muestran en las celdas. Esto quedará más claro con los ejemplos que analizaremos en este capítulo.

2 Tidy data

Antes de comenzar, recordemos establecer nuestro directorio de trabajo y las librerías que necesitamos para este capítulo:

```
setwd("~/Dropbox/Curso de R/Cap6_Datos_Ordenados")
options(scipen=999)
library (tidyverse)
library(readxl)
library(readr)
```

Recuerda que mediante el uso de la opción **options(scipen=999)** eliminamos la notación científica.

Tomemos de ejemplo las siguientes tablas o bases de datos del archivo de excel "tablas_ejemplo":

```
tabla_1 ← read_excel("tablas_ejemplo.xls", sheet="tabla1" )
tabla_2 ← read_excel("tablas_ejemplo.xls", sheet="tabla2" )
tabla_3 ← read_excel("tablas_ejemplo.xls", sheet="tabla3" )
```

Observa que al cargar la información con la función **read_excel()** hemos añadido la opción **sheet="-tabla1"** para indicar que nos interesa especificamente la hoja etiquetada en el excel como *tabla1*.

Comparemos la forma de presentar la información de estas tres tablas que acabamos de importar:

```
tabla_1
```

```
## # A tibble: 3 x 3
##   Nombre      `Tratamiento A`  `Tratamiento B`
##   <chr>            <dbl>            <dbl>
## 1 John Smith       NA               2
## 2 Jane Doe         16               11
## 3 Mary Johnson     3                1
```

```
tabla_2
```

```
## # A tibble: 2 x 4
##   Tratamiento        `John Smith`    `Jane Doe` `Mary Johnson`
##   <chr>            <dbl>            <dbl>      <dbl>
## 1 Tratamiento A    NA               16         3
## 2 Tratamiento B    2                11         1
```

```
tabla_3
```

```
## # A tibble: 6 x 3
##   Persona          Tratamiento           Resultado
##   <chr>            <chr>            <dbl>
## 1 John Smith       A                NA
## 2 Jane Doe         A                16
## 3 Mary Johnson     A                3
## 4 John Smith       B                2
## 5 Jane Doe         B                11
## 6 Mary Johnson     B                1
```

Las tres tablas muestran la misma información, pero ordenada de forma diferente. En la *tabla_1* cada fila representa un solo individuo, y cada columna el resultado de cierto tratamiento. La *ta-*

bla_2 muestra ahora en cada fila el tipo de tratamiento y cada columna representa a un individuo. Finalmente, la *tabla_3* reorganiza la información de tal manera que cada fila representa la combinación de cada individuo con cada tratamiento, mientras que en una sola columna se muestra el resultado de cada tratamiento, la *tabla_3* es un ejemplo de **tidy data.**

Los datos de forma *tidy data* son más fáciles de utilizar con la paquetería *tidyverse.* Además, las paqueterías *dplyr* y *ggplot2* también están diseñadas para trabajar con datos en esta misma forma.

Recordemos estas tres reglas para tener una base de datos del tipo *tidy*:

- Cada fila representa un individuo u observación
- Cada columna representa una variable
- Cada celda representa un valor

Además, una tabla debe tener una misma unidad de observación, es decir, si estamos midiendo variables de individuos, no podemos combinarla con hogares o empresas. Estos últimos deben tener su propia tabla. Esto tipo de tablas o bases de datos, se trabajarán en el siguiente capítulo.

Anteriormente trabajamos la base de datos *enigh*, ¿dirías que es *tidy data*?, ¿porqué?

R: Sí, porque cada fila representa a un individuo u observación (en este caso hogares), y cada columnas representa el resultado de la información obtenida en cada variables, las celdas representa los valores de cada hogar.

En lo que sigue de este capítulo aprenderemos las principales funciones que nos permiten acomodar los datos para que sean del tipo *tidy data*.

3 Spreading y Gathering

Desafortunadamente en muchas ocasiones los datos que obtengamos no tendrán la forma de tidy data, por lo que necesitamos no solo manipular los datos para generar nuevas variables como lo hemos hecho anteriormente, ademas será necesario efectuar primero cierta manipulación para darles la forma de tidy data.

Dos razones por las que generalmente no tendrás datos en forma tidy data son:

- La mayoría de las personas no están familiarizadas con este tipo de estructura.
- Las bases de datos son comúnmente organizadas para facilitar otro tipo de tareas distintas al análisis, por ejemplo: facilitar la captura o recolección de los datos.

Para transformar nuestros datos a la forma tidy data¨, es necesario identificar cuales son las observaciones y cuales las variables, generalmente esta es una tarea sencilla, aunque en ocasiones también implica consultar documentación sobre la base de datos. Paso seguido es necesario resolver uno de dos problemas comunes:

- Una variable puede estar dispersa a través de múltiples columnas
- Una observación puede estar dispersa a través de múltiples filas

Para solucionar estos dos problemas necesitaremos dos funciones incluidas en tidyr, estas son gather() y spread().

3.1 Gather()

Para ejemplificar esta función, importemos los datos de algunos países sobre PIB (Producto Interno Bruto),

población y cajeros automáticos desde 1960 a 2020 obtenidos del "Banco Mundial". Esta información se

encuentra contenida en el csv, de nombre datos_banco_mundial.

```
datos <- read_csv("datos_banco_mundial.csv")
##
## -- Column specification ----------------------------------------------------
----------------------------
## cols(
##   .default = col_character(),
##   `2007` = col_double(),
##   `2008` = col_double(),
##   `2009` = col_double()
## )
## i Use `spec()` for the full column specifications.
```

```
dim(datos)
## [1] 21 65
```

```
datos[1:6,1:3]
## # A tibble: 6 x 3
##   `Pais nombre`    ` Pais    `serie Name`
##   <chr>            <chr>     <chr>
## 1 Argentina        ARG       PIB (US$ a precios constantes de 2010)
## 2 Argentina        ARG       Poblacion, total
## 3 Argentina        ARG       Cajeros automaticos (por cada 100.000 adul-
tos)
## 4 Canadá           CAN       PIB (US$ a precios constantes de 2010)
## 5 Canadá           CAN       Poblacion, total
## 6 Canadá           CAN       Cajeros automaticos (por cada 100.000 adul-
tos)
```

La tabla contiene 21 filas y 65 columnas.

En esta tabla primero debemos identificar la unidad de observación, cuales son las observaciones y cuales las variables. Tras una revisión podremos darnos cuenta qué la unidad de observación son países, y que para cada país tenemos tres variables (PIB, población y cajeros automáticos). La revisión también nos muestra que cada una de estas variables se identifica por su valor a través del tiempo, desde 1960 a 2020. Evidentemente esta tabla de datos no es del tipo tidy data, las variables están en filas y no en columnas y los años están en columnas en vez de filas.

Para cada año existe una columna que indica los valores de estas tres variables para los diferentes países.

Los años en si, no son variables, sino más bien valores, por lo que no deben ser nombres de columnas. Para resolver esto debemos usar la función *gather()*, la cual transformará esas columnas a filas:

```
#Observa lo que sucede:
datos ← datos%>%
     gather(`1960` : `2020`, key="periodo", value = "valores")
dim(datos)
```

```
## [1]  1281     6
```

```
datos[1:6,1:6]
```

```
## # A tibble: 6 x 6
## `Pais nombre `Pais `serie Name`           `serie Code`     periodo valores
##    <chr>      <chr>  <chr>                         <chr>       <chr>   <chr>
## 1 Argentina  ARG   PIB (US$ a precios constant~ NY.GDP.MKTP.~ 1960 1.16E+~
## 2 Argentina  ARG   Poblacion, total       SP.POP.TOTL      1960    204817~
## 3 Argentina  ARG   Cajeros automaticos (por ca~ FB.ATM.TOTL.~ 1960    ..
## 4 Canadá      CAN   PIB (US$ a precios constant~ NY.GDP.MKTP.~ 1960 2.94E+~
## 5 Canadá      CAN Poblacion, total        SP.POP.TOTL      1960    179090~
## 6 Canadá      CAN Cajeros automaticos (por ca~ FB.ATM.TOTL.~ 1960     ..
```

Observa que en este caso en el nombre de columnas *1960 y 1920* se han incluido estos caracteres (1960:2020). Esto sucede porque no es común que una columna tenga un nombre numérico. Para que **R** lo entienda debemos acompañar el nombre de la variable con estas pequeñas comillas. Lo mismo sucede con los nombres de columna que tienen algun espacio en blanco.

La instrucción anterior tomó las columnas llamadas *"1960"* hasta *"2020"* y generó dos variables. La primera llamada "periodo" la cual ahora contiene los nombres de las columnas, es decir, los años de 1960 hasta 2020. A la variable que tome los nombres de las columnas se debe poner siempre en la opción *key*.

La segunda variable es "valores" la cual tiene el valor de las variables. Este nombre siempre se va a especificar en la opción "value". Los valores ahora se encuentran apiladas hacia abajo en vez de en columnas.

Observa también que ahora tenemos 1286 filas y 6 columnas.

3.2 Spreading()

La función *gather()* nos permite reunir o apilar columnas en filas ordenadas por una variable clave, en estecaso, el periodo o años. Sin embargo, nuestros datos aún no son del tipo *tidy data*, las variables (PIB, población total y cajeros) deben ser columnas, es decir, tenemos tres variables y necesitamos tener una columna para cada variable. Actualmente éstas se encuentran en filas. Lo que necesitamos ahora es una función "contraria" a la función *gather()*, esta función opuesta se llama *spreading()*

```
datos ← datos %>%
     select(-(`serie Code`)) %>%
spread(key = "serie Name", value="valores")
dim(datos)
```

```
## [1] 427 6
```

```
datos[1:10,]
```

```
## # A tibble: 10 x 6
##    `Pais nombre` Pais    periodo `Cajeros automa~ `PIB (US$ a pre~
##    <chr>        <chr>     <chr>      <chr>        <chr>
## 1  Argentina   ARG        1960       ..           1.16E+11
## 2  Argentina   ARG        1961       ..           1.22E+11
## 3  Argentina   ARG        1962       ..           1.21E+11
## 4  Argentina   ARG        1963       ..           1.14E+11
## 5  Argentina   ARG        1964       ..           1.26E+11
## 6  Argentina   ARG        1965       ..           1.39E+11
## 7  Argentina   ARG        1966       ..           1.38E+11
## 8  Argentina   ARG        1967       ..           1.43E+11
## 9  Argentina   ARG        1968       ..           1.50E+11
## 10 Argentina   ARG        1969       ..           1.64E+11
## # ... with 1 more variable: `Poblacion, total` <chr>
```

Observa que primero hemos efectuado una selección de la columna *serie Code*. Paso seguido hemos indicado una nueva **key="serie Name"**, gracias a la cual estamos indicando que se generarán tantas variables como valores diferentes existan en esta columna. Finalmente, indicamos que en

cada caso el valor de la observación correspondiente a cada variable, está contenido en la columna *valores*. De ahí que la indicación **sea value="valores"**

Nuestros datos son ahora *tidy data*. Las filas representan a cada país en cada año y cada columna es una variable. Sin embargo nuestro trabajo aún no termina, las tres variables son identificadas como texto, ya que contienen los signos ".." que indica valor perdido. Necesitamos realizar dos tareas más para tener una base de datos lista para analizar. La primera es reemplazar el texto ".." por valores perdidos y la segunda reemplazarlas tres variables como variables numéricas.

3.2.1 Valores perdidos

Para lidiar con los valores perdidos aplicamos las técnicas que hemos estudiado anteriormente. Primero hacemos que en todos los casos donde se identifiquen ".." se reemplacen estos valores por *NA*. Esto para las tres variables que ahora forman parte de la base de datos.

```
datos ← datos %>%
     mutate(`Cajeros automaticos (por cada 100.000 adultos)` = ifelse
      (`Cajeros automaticos (por cada 100.000 adultos)`=".." , NA,
      `Cajeros automaticos (por cada 100.000 adultos)`) ) %>%
     mutate(`PIB (US$ a precios constantes de 2010)` = ifelse
      (`PIB (US$ a precios constantes de 2010)`=".." , NA,
      `PIB (US$ a precios constantes de 2010)`) ) %>%
     mutate(`Poblacion, total` = ifelse
      (`Poblacion, total`=".." , NA,
      `Poblacion, total`) )
```

Observa que en todos los casos el nombre de la variable esta acompañado de los pequeñas comillas (acento grave), la razón de ello es la que comentamos con anterioridad.

Si exploras brevemente el contenido dentro de datos te darás cuenta que ahora, se incluyen los valores perdidos como *NA*.

3.2.2 Tipos de variables

Al obtener un resumen de nuestra nueva base de datos, observamos lo siguiente;

```
summary(datos)
```

```
##    Pais nombre          Pais             periodo
##    Length:427           Length:427       Length:427
##    Class :character     Class :character Class :character
##    Mode  :character     Mode  :character  Mode  :character
##    Cajeros automaticos (por cada 100.000 adultos)
##    Length:427
```



```
##    Class :character
##    Mode  :character
##    PIB (US$ a precios constantes de 2010) Poblacion, total
##    Length:427                             Length:427
##    Class :character                       Class :character
##    Mode  :character                       Mode  :character
```

Esto nos indica que las nuevas variables que hemos construido son de tipo carácter y no numérico. Esto limitará cualquier análisis que deseemos efectuar. Debemos entonces corregir esta irregularidad, para que nuestros datos sean del tipo correcto.

Para ello usamos la función **as.numeric();**

```
datos ← datos %>%
      mutate(periodo=as.numeric(periodo)) %>%
      mutate(`Cajeros automaticos (por cada 100.000 adultos)`=as.numeric
           (`Cajeros automaticos (por cada 100.000 adultos)`)) %>%
      mutate(`PIB (US$ a precios constantes de 2010)`=as.numeric
           (`PIB (US$ a precios constantes de 2010)`)) %>%
      mutate(`Poblacion, total`=as.numeric
           (`Poblacion, total`))
```

Observa que en todos los casos hemos efectuado un reemplazo de la variable actual, por una nueva variable que tiene el mismo nombre, pero que será del tipo numérico. Si solicitamos nuevamente un resumen de *datos* veremos que se ha cambiado el tipo de variables. Ahora conocemos los estadísticos básicos de cada una de ellas y podemos ver los valores *NA* que existen en cada caso.

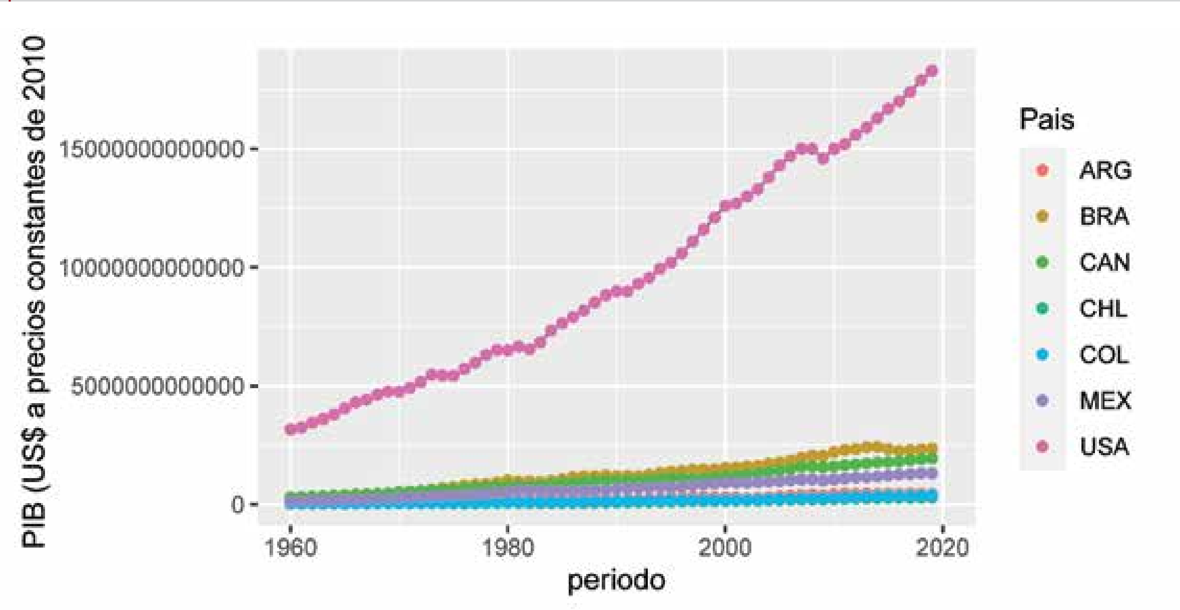
```
summary(datos)
```

```
##    Pais nombre          Pais             periodo
##    Length:427           Length:427       Min.   :1960
##    Class :character     Class :character 1st Qu.:1975
##    Mode  :character     Mode  :character Median :1990
##                                          Mean   :1990
##                                          3rd Qu.:2005
##                                          Max.   :2020
##
##    Cajeros automaticos (por cada 100.000 adultos)
```

```
##    Min. : 1.871
##    1st Qu.: 41.372
##    Median : 55.620
##    Mean : 88.483
##    3rd Qu.:117.120
##    Max. :228.429
##    NA's :328
##    PIB (US$ a precios constantes de 2010) Poblacion, total
##    Min. : 29375692257                  Min. : 8132990
##    1st Qu.: 193750000000               1st Qu.: 24039927
##    Median : 445000000000               Median : 39657130
##    Mean : 1874444953650                Mean : 83935403
##    3rd Qu.: 1287500000000              3rd Qu.:120439848
##    Max. :18300000000000                Max. :328239523
##    NA's :7                             NA's :7
```

Finalmente ahora que nuestros datos son del tipo *tidy data*¨ y que hemos preparado nuestra base de datos, podemos proceder a realizar algún análisis requerido. En este caso consideremos simplemente una gráfica en la que podamos ver la evolución del PIB de cada país;

```
ggplot(datos, aes(periodo, `PIB (US$ a precios constantes de 2010)`))+
     geom_line(aes(group = Pais), color = "grey50", na.rm = TRUE) +
geom_point(aes(color = Pais), na.rm = TRUE)
```



4 Separar y unir

Ahora utilizaremos dos funciones útiles cuando nuestros datos contienen variables que están combinadas en una sola columna, o lo contrario, una variable que esta separada en varias columnas. Para este caso trabajaremos con la lista de Investigadores del Sistema Nacional de Investigadores de México vigentes en 2018. Este archivo es público y está disponible en Internet.

```
sni ← read_csv("Investigadores-SNI-Vigentes-2018.csv", )
```

```
##
## -- Column specification --------------------------------------------------
----------------------------
## cols(
##   Grado = col_character(),
##   `Apellido Paterno` = col_character(),
##   `Apellido Materno` = col_character(),
##   Nombre = col_character(),
##   Nivel = col_character(),
##   `Institucion de Adscripcion` = col_character(),
##   `Area del Conocimiento` = col_character()
## )
```

```
sni
```

```
## # A tibble: 28,578 x 7
##   Grado `Apellido Pater~ `Apellido Mater~ Nombre    Nivel       `Institu-
cion de~
##   <chr> <chr>            <chr>      <chr>      <chr> <chr>
## 1 DRA.  HERNANDEZ        LOPEZ      SILVI~     C     AGROPECUARIA EL~
## 2 DR.   FINKELSTEIN      SHAPIRO    DANIEL     1     ARIZONA STATE U~
## 3 DR.   ALEJO            JAIME      ANTON~     1     ASOCIACION MEXI~
## 4 MED.  GIL              CARRASCO   FELIX      1     ASOCIACION PARA~
## 5 M. E~ RUIZ             CRUZ       MATIL~     1     ASOCIACION PARA~
## 6 MED.  JIMENEZ          ROMAN      JESUS      1     ASOCIACION PARA~
## 7 DR.   HERNANDEZ        ZIMBRON    LUIS ~     1     ASOCIACION PARA~
## 8 DR.   FROMOW           GUERRA     JORGE~     1     ASOCIACION PARA~
## 9 MED.  QUIROZ           MERCADO    HUGO       3     ASOCIACION PARA~
## 10 DR.  BENAVIDES        PERALES    GUILL~     2     BANCO DE MEXICO
## # ... with 28,568 more rows, and 1 more variable: `Area del Conocimiento`
<chr>
```

Al observar los datos notamos dos cosas: primero, el nombre de los investigadores se divide en tres columnas. Segundo, el Área del conocimiento del investigador se encuentra en una sola columna la clave y el significado de cada área. Dado que no nos interesa conocer los nombres de los investigadores, este puede estar en una sola columna y no tres, en cambio el área del conocimiento debe estar separado una columna para la clave y otra columna para el significado de cada clave.

Usaremos unir y separar para tener la base deseada.

4.1 Unir

Para juntar el contenido de varias columnas en una sola necesitamos utilizar la función *unite()*, en este caso juntaremos el nombre de los investigadores en una solo columna:

```
sni ← sni %>%
     unite(Nombre, `Apellido Paterno`, `Apellido Materno`, Nombre, sep=" ")
```

La sintaxis para usar esta función es; primero indicar el nombre de la nueva variable, donde se encontrará el contenido de las columnas que deseamos unir. Paso seguido se indica, separado por comas, los nombres de las variables a unir, y finalmente se utiliza la opción "sep", la cual sirve para indicar como deseamos que se identifique la separación de las variables unidas dentro de la misma columna. De manera predeterminada la función utiliza un guión bajo. Al utilizar la opción *sep*=" " se muestran los apellidos y nombre separados por un espacio blanco entre ellos.

4.2 Separar

La función *separate()* es el contrario la función unir, la cual permite dividir el contenido de una columna en varias, dependiendo de un carácter especial, este puede ser un espacio, un guion, un punto, etc. En nuestro ejemplo, deseamos separa el área del conocimiento en dos, una columna que tenga la clave o número del área y otra que tenga el nombre completo del área del conocimiento:

```
sni ← sni %>%
     separate(`Area del Conocimiento`, into = c("cve_area", "area"), sep =
":")
```

La instrucción comienza diciendo que columna es la que deseamos separar. La indicación **into = c("cve_area", "area")** señala cuales son las nuevas variables que deseamos obtener después de la separación y la opción **sep = ":"** indica cual es el caractér que nos indica a partir de donde se debe partir el texto.

5 Actividades

1. Utilizando el data frame datos ya en su forma tidy data, construye las siguientes variables:

- Logaritmo natural del PIB
- PIB per cápita
- Logaritmo del PIB per cápita
- Logaritmo de los Cajeros automáticos (por cada 100.000 adultos)
- Gráfica un diagrama de dispersión entre el logaritmo del PIB per cápita y el logaritmo del los cajeros automáticos.

2. Importa la tabla "importaciones_exportaciones"

- Realiza la manipulación necesaria para transformar estos datos a tipo *tidy data*
- Gráfica el logaritmo del las exportaciones y las importaciones
- Calcula la diferencia (déficit comercial) entre las exportaciones e importaciones
- Gráfica el logaritmo del déficit comercial

Capítulo 7

Datos Relacionales

Capítulo 7 | Datos Relacionales

1 Introducción

Hasta ahora hemos trabajado con datos que se encuentran contenidos dentro de una misma base. En ocasiones es necesario trabajar con datos que se encuentran en diferentes bases. Por ejemplo, en el capítulo 4, usamos la base de datos de la ENIGH. Esta base contiene información concentrada de las características de los hogares en México. Si trabajamos únicamente con ella, decimos que se trata de una base con datos ordenados, como lo vimos en el capítulo anterior. Cada hogar se compone de diversos integrantes (personas), que tienen características específicas, como edad, sexo, nivel educativo etc. Imagina ahora que deseemos conocer cual es la edad promedio de los integrantes del hogar en cada entidad federativa, o que deseemos conocer el promedio de años de escolaridad de los miembros de un hogar. En este caso deberíamos hacer uso de dos bases de datos; una donde se encuentra la información general del hogar y otra donde se encuentra la información de cada uno de los individuos que conforman el hogar. Este tipo de datos se conoce como **relacionales,** precisamente porque es necesario establecer relaciones entre ellos para poder analizarlos. En este capítulo nos enfocaremos en el manejo de este tipo de datos, lo cual implica el uso de mas de una fuente de información. Para ello usaremos tres tipos de acciones;

- Uniones de transformación (mutating joins) es el procedimiento de agregar nuevas variables a un **data frame**, las cuales provienen de otro **data frame**, bajo la condición de que existan observaciones coincidentes entre ambas.
- Uniones de filtro (filtering joins) es el procedimiento de filtrar observaciones en un **data frame dependiendo** de si hay o no coincidencia con una observación de otro data frame.
- Operaciones de conjuntos (set operations) es el procedimiento de tratar las observaciones como elementos de un conjunto.

1.1 Previos

En este capítulo trabajaremos con 5 bases de datos que se relacionan entre si. En este caso nuestro archivos se encuentran en un formato de datos conocido como **.dta**. Por ello debes instalar la librería **haven**, por medio la cual podremos importar archivos que provengan de fuentes como Stata, SPSS y SAS, programas especializados en el manejo de datos, principalmente provenientes de encuestas. Recuerda que, para cargar tus datos, debes indicar el directorio donde encuentran las bases.

```
setwd("~/Dropbox/Curso de R/Cap7_Datos_Relacionales")
library(haven)
library(tidyverse)
hogares ← read_dta("hog_jal.dta")
viviendas ← read_dta("viv_jal.dta")
personas ← read_dta("per_jal.dta")
trabajos ← read_dta("trab_jal.dta")
ingresos ← read_dta("ing_jal.dta")
```

En este caso trabajaremos únicamente con los datos del estado de Jalisco.

2 Sobre la ENIGH

La ENIGH tiene como objetivo medir los patrones de ingreso y gasto en México. Hasta ahora hemos usado la ENIGH a través del uso únicamente de información que se encuentra desagregada a nivel de hogares. Sin embargo, cada hogar se compone de personas con diferentes características. Cada persona puede tener uno o mas empleos, con determinas prestaciones, por ejemplo, el pago de aguinaldo, vacaciones o utilidades. Además, en cada trabajo, las personas reciben diferentes ingresos por cada uno de los diferentes conceptos. Toda esta información se encuentra en la ENIGH. Como te imaginarás es demasiada información y no es posible que toda exista en la misma base de datos, pues cada tipo de información se encuentra desagregada a diferentes niveles de estudio.

La ENGIH también incluye las características de la vivienda y considera que dentro de una misma vivienda pueden vivir uno o mas hogares. Por ejemplo, considera una familia de mamá, papá y dos hijos. Si uno de los hijos está casado y viven en la misma vivienda con su esposa, entonces se considera que en la vivienda hay dos hogares. El primero formado por mamá, papá y el hijo soltero. El segundo hogar formado por el hijo casado y su esposa. En este caso se contabilizará una vivienda, cuyas características físicas se incluyen en la base de datos **vivienda**, dos hogares, cuya características están en la base de datos **hogares** y cinco personas, con su información sociodemográfica correspondiente en la base de datos **personas**.

Además, cada persona puede tener cero, uno o más empleos. La información sobre las características de su empleo se incluye en la base de datos de **trabajo**. Finalmente, los ingresos por cada uno de los diferentes conceptos (utilidades, vacaciones, aguinaldo, etc.) se incluyen en la base de datos **ingresos**.

Cada una de estas bases se relaciona con las demás. Para poder relacionarlas es necesario conocer el tipo de relación que se da entre ellas.

Debemos recalcar que en este capítulo usaremos únicamente cinco bases de datos de la ENIGH, la cual es mucho mas compleja y contiene por lo menos cuatro bases de datos mas, relativas a los patrones de consumo individuales. En este ejercicio decidimos incluir únicamente cinco, para mostrar las diferentes herramientas con las que contamos en **R**, para relacionarlas y analizarlas.

Comenzaremos explorando las características de cada una de estas bases;

La base de viviendas se compone de 8 columnas y para el caso de Jalisco se incluyen 2,095 observaciones. Cada observación en esta base corresponde a las características de una vivienda.

```
viviendas
```

```
## # A tibble: 2,095 x 8
## folioviv          tipo_viv dotac_agua disp_elect tenencia ubica_geo est_socio edo
##    <chr>      <chr> <chr> <chr> <chr>     <chr>     <chr>  <chr>
## 1  1400180001 1     1     1     4         14039     3      14
## 2  1400180002 1     1     1     4         14039     3      14
## 3  1400180005 1     1     1     1         14039     3      14
## 4  1400180006 1     1     1     4         14039     3      14
## 5  1400189601 1     1     1     1         14039     3      14
## 6  1400189603 2     1     1     1         14039     3      14
## 7  1400189604 1     1     1     4         14039     3      14
## 8  1400189605 2     1     1     1         14039     3      14
## 9  1400189606 1     1     1     4         14039     3      14
## 10 1400264701 1     1     1     1         14039     3      14
## # ... with 2,085 more rows
```

Cada observación en la base de datos de hogares corresponde a un hogar. Observa que hay mas hogares que viviendas, precisamente porque en una vivienda puede haber mas de un hogar.

```
hogares
```

```
## # A tibble: 2,152 x 9
##    folioviv   foliohog ubica_geo clase_hog ing_cor gasto_mon alimentos salud
##    <chr>      <chr> <chr>      <chr> <dbl>      <dbl>      <dbl>  <dbl>
## 1  1400180~   1     14039      2     28869.     16068.     8511.      0
## 2  1400180~   1     14039      3     57237.     32342.     12793.     0
## 3  1400180~   1     14039      1     32527.     24481.     9148.      0
## 4  1400180~   1     14039      1     28381.     17468.     7579.      0
## 5  1400189~   1     14039      2     39775.     70465.     21188.     0
## 6  1400189~   1     14039      2     102717.    56377.     13731.     3073.
## 7  1400189~   1     14039      2     91475.     43816.     12568.     154.
## 8  1400189~   1     14039      2     55828.     60636.     15300.     0
## 9  1400189~   1     14039      2     40313.     12303.     5901.      1027.
## 10 1400189~   2     14039      2     23390.     21947.     14374.     318.
## # ... with 2,142 more rows, and 1 more variable: educa_espa <dbl>
```

En la base personas, cada observación corresponde a determinadas características de una persona. Observa que hay muchas mas personas que hogares, porque en un hogar viven 1 o una o más personas.

```
personas
```

```
## # A tibble: 7,769 x 9
##    folioviv        foliohog numren sexo edad disc1 hablaind alfabetism asis_esc
##    <chr>           <chr> <chr> <chr> <dbl>     <chr> <chr>   <chr>  <chr>
## 1  1400180001      1     01    1     81        2     2             1    2
## 2  1400180001      1     02    2     73        1     2             1    2
## 3  1400180002      1     01    1     78        2     2             1    2
## 4  1400180002      1     02    2     70        4     2             1    2
## 5  1400180002      1     03    2     50        8     2             1    2
## 6  1400180002      1     04    1     41        8     2             1    2
## 7  1400180002      1     05    2     8         8     2             1    1
## 8  1400180005      1     01    1     35        8     2             1    2
## 9  1400180006      1     01    2     54        8     2             1    2
## 10 1400189601      1     01    1     46        8     2             1    2
## # ... with 7,759 more rows
```

La base de datos trabajo, contiene información sobre el tipo de trabajo y las prestaciones que ofrece. Las personas pueden tener 0, 1 o más empleos. Observa que en esta base se incluye una variable que denomina numren que otorga un número a cada persona del hogar.

```
trabajos
```

```
## # A tibble: 4,205 x 7
##    foliviv         foliohog   numren     id_trabajo  pres_2 pres_3 pres_4
##    <chr>       <chr>      <chr>      <chr>      <chr>  <chr>  <chr>
## 1  1400180002 1          01         1          ""     ""     ""
## 2  1400180002 1          04         1          ""     ""     ""
## 3  1400180005 1          01         1          "02"   "03"   "04"
## 4  1400180006 1          01         1          "02"   "03"   ""
## 5  1400189601 1          01         1          ""     ""     ""
## 6  1400189601 1          02         1          "02"   "03"   ""
## 7  1400189601 1          04         1          ""     ""     ""
## 8  1400189603 1          02         1          "02"   "03"   ""
## 9  1400189603 1          02         2          "02"   "03"   ""
## 10 1400189604 1          01         1          "02"   "03"   ""
## # ... with 4,195 more rows
```

Finalmente, la base de datos de ingresos, indica los ingresos por trabajo, haciendo una clasificación por conceptos. Cada observación en esta base se refiere a un concepto de ingreso para determinada persona.

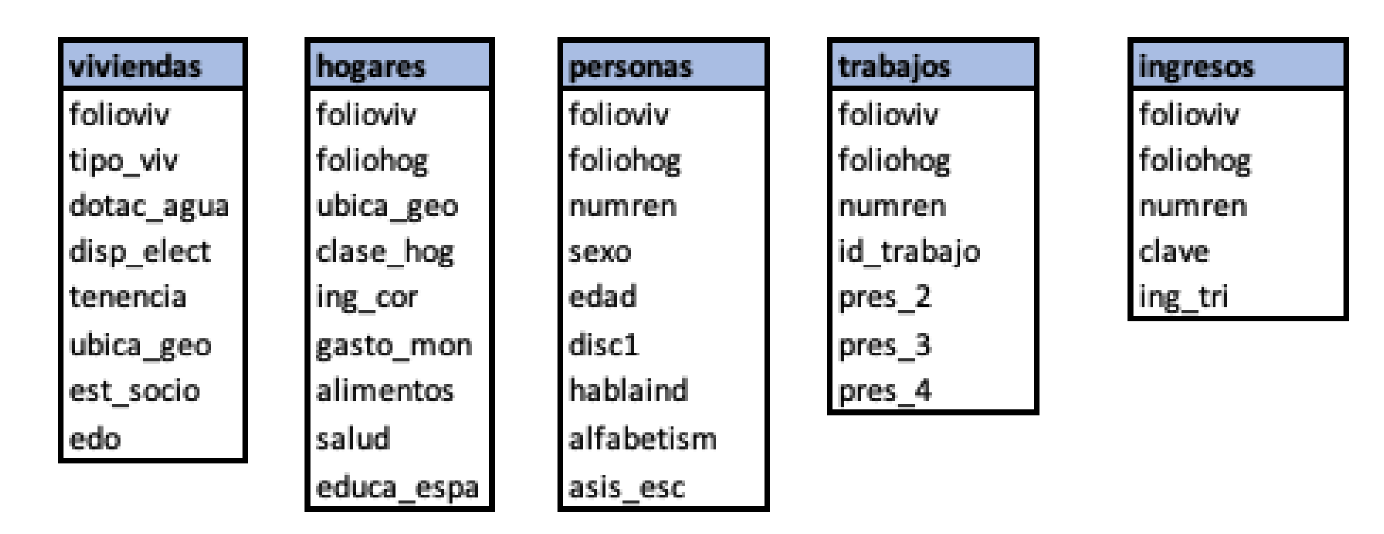
```
ingresos
```

```
## # A tibble: 10,919 x 5
##    foliviv    foliohog   numren     clave      ing_tri
##    <chr>      <chr>      <chr>      <chr>      <dbl>
## 1  1400180001 1          02         P044       1702.
## 2  1400180001 1          02         P040       3228.
## 3  1400180001 1          01         P032       6457.
## 4  1400180001 1          01         P023       5870.
## 5  1400180002 1          01         P024       1761.
## 6  1400180002 1          01         P044       1702.
## 7  1400180002 1          02         P040       1908.
## 8  1400180002 1          02         P044       1702.
## 9  1400180002 1          04         P001       23478.
```

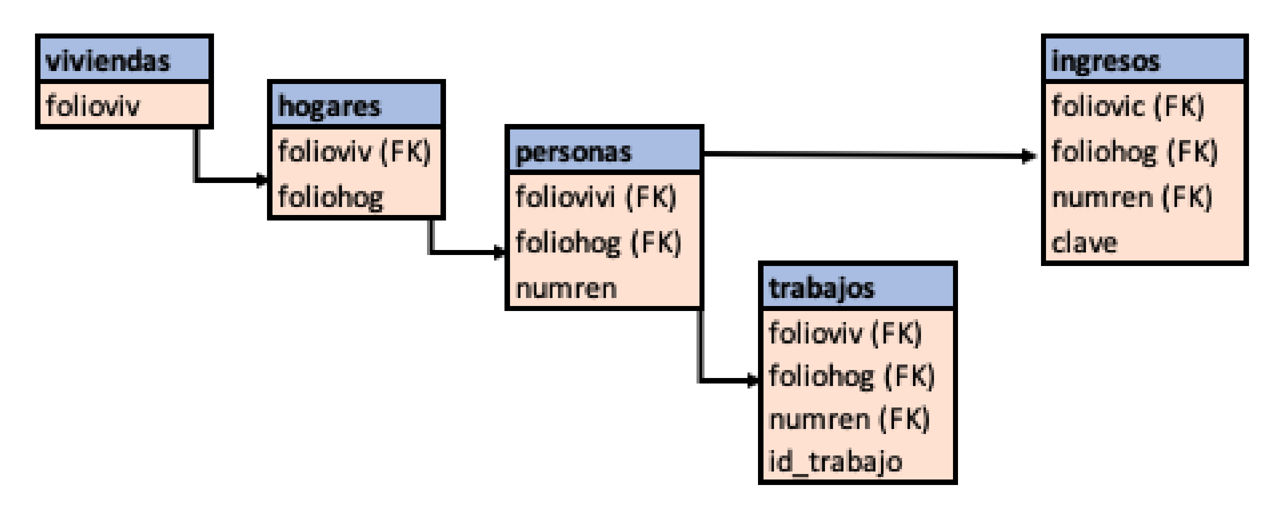


```
## 10 1400180002 1          01         P003       7435.
## # ... with 10,909 more rows
```

De esta manera vemos que las variables contenidas en cada una de las bases son;



Las diferentes relaciones que se establecen entre ellas se muestran en el siguiente diagrama



Es decir, podemos relacionar la base de datos de viviendas con hogares, esta a su vez con la de personas. La base de datos de personas se puede relacionar con la base de ingresos y la base de trabajos. Sin embargo, la base de datos de trabajo no se puede relacionar con la base de manera directa con los ingresos. Esto significa que para cada persona podemos saber su ingreso, pero no podemos saber sobre cual de sus empleos proviene exactamente ese nivel de ingreso. Las variables incluidas en este diagrama son las variables principales que determinan las relaciones entre las diferentes bases, la unión de ellas genera la clave primaria dentro de cada base. En algunos casos se ha añadido la indicación (FK) lo cual indica que se trata de una llave o clave foránea. En el siguiente apartado explicamos a que se refieren estos términos..

3 Llaves de relación (Keys)

La o las variables que se usan para conectar cada par de bases de datos, se conocen como llaves. En los datos relacionales se identifican dos tipos de llaves; foráneas y primarias.

- Llaves primarias: Son aquellas que identifican una observación dentro de su propia base de datos. Por ejemplo, la variable *numren* identifica a cada persona dentro de la base de datos **personas**, mientras que la variable *clave* identifica el concepto del ingreso por trabajo en la base **ingresos.**
- Llaves foráneas: Son aquellas que identifican una observación fuera de su base de datos. Por ejemplo, la clave **foliviv** de la base hogares, es una llave foránea porque gracias a ella podemos identificar a cada hogar en una vivienda.

Una llave puede ser tanto foránea como primaria. Por ejemplo, la variable *foliohog* dentro de la base de hogares es primaria, además es foránea porque nos permite identificar a cada persona, cada trabajo y cada ingreso en un hogar determinado.

Una vez que hemos identificado las llaves que son primarias, debemos asegurarnos qué efectivamente identifiquen a una sola observación. Veamos el caso de la base de datos **viviendas**.

```
viviendas %>%
     count(folioviv)
```

```
## # A tibble: 2,095 x 2
##    folioviv      n
##    <chr>         <int>
## 1  1400180001    1
## 2  1400180002    1
## 3  1400180005    1
## 4  1400180006    1
## 5  1400189601    1
## 6  1400189603    1
## 7  1400189604    1
## 8  1400189605    1
## 9  1400189606    1
## 10 1400264701    1
## # ... with 2,085 more rows
```

Observa que hemos solicitado que se efectué primero un conteo de la variable *folioviv*. Ahora filtremos cuales de esos folios aparecen mas de una vez.


```
viviendas %>%
     count(folioviv) %>%
     filter(n>1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: folioviv <chr>, n <int>
```

Obtenemos que hay cero folios que cumplen esa condición, por lo cual la variable *folioviv*, identifica de manera única a cada vivienda.

En ocasiones, la variable que define una llave primaria va acompañada de otras variables. Por ejemplo, en el caso de la base de datos hogares. La variable llave en este caso es foliohog.

```
hogares %>%
     count(foliohog) %>%
     filter(n>1)
```

```
## # A tibble: 4 x 2
##    foliohog   n
##    <chr>      <int>
## 1  1          2095
## 2  2          50
## 3  3          5
## 4  4          2
```

En este caso observamos que la variable toma valores de 1,2,3 y 4.

Esto es porque la llave en esta base se conforma con dos variables, *folioviv* y *foliohog* (como se observó en el diagrama anterior). Para verificarlo hagamos el conteo con ambas variables

```
hogares %>%
count(folioviv, foliohog) %>%
filter(n>1)
```

```
## # A tibble: 0 x 3
## # ... with 3 variables: folioviv <chr>, foliohog <chr>, n <int>
```

Identifiquemos la composición de la llave primaria del resto de bases de datos. Para las personas tenemos;

```
personas %>%
     count(folioviv, foliohog, numren) %>%
     filter(n>1)
```

```
## # A tibble: 0 x 4
## # ... with 4 variables: folioviv <chr>, foliohog <chr>, numren <chr>, n <int>
```

Mientras que para los trabajos

```
trabajos %>%
     count(folioviv, foliohog, numren, id_trabajo) %>%
     filter(n>1)
```

```
## # A tibble: 0 x 5
## # ... with 5 variables: folioviv <chr>, foliohog <chr>, numren <chr>,
## # id_trabajo <chr>, n <int>
```

Finalmente, para los ingresos

```
ingresos %>%
     count(folioviv, foliohog, numren, clave) %>%
     filter(n>1)
```

```
## # A tibble: 0 x 5
## # ... with 5 variables: folioviv <chr>, foliohog <chr>, numren <chr>,
## # clave <chr>, n <int>
```

Las relaciones entre las diferentes bases, así como la identificación de las llaves primarias y foráneas, suele ser parte de la documentación de una base de datos. En caso de no serlo, puedes efectuar un conteo como el que efectuamos para identificarlas.

4 Uniones de transformación

4.1 Los tipos de uniones

Antes de proceder a efectuar uniones con la ENIGH, profundicemos un poco, sobre como funcionan las uniones. Para ello crearemos los siguientes data frame.

llave	sexo	edad
1	H	25
2	M	40
3	H	18

llave	ingreso	gasto
1	1750	1100
2	2540	1760
3	3000	890

Los cuales se construir usando el siguiente código

```
x ← tribble(
     ~llave, ~sexo, ~edad,
     1, "H", "25",
     2, "M", "40",
     3, "H", "18"
)
```



```
y ← tribble(
     ~llave, ~ingreso, ~gasto,
     1, "1750", "1100",
     2, "2540", "1760",
     4, "3000", "890"
)
```

Existen diferentes maneras de relacionar o combinar estas dos pequeñas bases.

4.2 Unión interior

Una unión interior en la cual la nueva base resultante únicamente conservará las observaciones que este incluidas en ambas bases. Por ejemplo, en este caso las observaciones con clave 3 y 4 no se encuentran presentes en ambas bases, por lo cual se eliminan de la unión. En una unión interior estamos sujetos a la pérdida de observaciones. Las uniones interiores se efectuarán con la función **inner_join**

```
inner_join(x,y, by = "llave")
```

```
## # A tibble: 2 x 5
##    llave sexo      edad  ingreso    gasto
##    <dbl> <chr>     <chr> <chr>      <chr>
## 1  1     H         25    1750       1100
## 2  2     M         40    2540       1760
```

4.3 Unión exterior

A diferencia de las uniones interiores, una unión exterior conserva observaciones siempre y cuando se encuentren presente en alguna de las dos bases. Es decir, no exige su presencia en ambas.

Existen tres tipos de uniones exteriores

- Left join: Esta unión conservará las observaciones que se encuentren en la base que este a la izquierda. Si no existe su correspondiente observación en la base de la derecha asignará un NA.

```
left_join(x,y, by="llave")
```

```
## # A tibble: 3 x 5
##    llave sexo      edad  ingreso    gasto
##    <dbl> <chr>     <chr> <chr>      <chr>
## 1  1     H         25    1750       1100
## 2  2     M         40    2540       1760
## 3  3     H         18    <NA>       <NA>
```

En este caso, debido a que la observación con clave 3 se encuentra únicamente en x y x se declaró a la izquierda de y, se conservará la observación de x, asignado un NA a su correspondiente valor de la variable en y.

- Right join: Esta unión funciona de manera semejante que **left_join** con la diferencia que conservará las observaciones que este en la base de la derecha y asignará un correspondiente NA a la variable de la base en la izquierda.

```
right_join(x,y, by="llave")
```

```
## # A tibble: 3 x 5
##   llave sexo edad ingreso   gasto
##   <dbl> <chr> <chr> <chr>    <chr>
## 1 1     H     25    1750     1100
## 2 2     M     40    2540     1760
## 3 4     <NA>  <NA>  3000     890
```

- Full join: Esta unión conserva todas las observaciones.

```
full_join(x,y, by="llave")
```

```
## # A tibble: 4 x 5
## llave   sexo edad ingreso    gasto
## <dbl>   <chr> <chr> <chr>    <chr>
## 1 1     H     25    1750     1100
## 2 2     M     40    2540     1760
## 3 3     H     18    <NA>     <NA>
## 4 4     <NA> <NA>   3000     890
```

La decisión sobre que tipo de unión deseamos utilizar dependerá de las necesidades que tengamos y cual sea el análisis que vamos a efectuar.

Observa que en todos los casos hemos indicado **by="llave"** para indicar que es la variable común a ambas bases de datos. Si no indicamos esta **llave** de forma automática **R** procederá a buscar variables comunes en ambas bases. Si las encuentra asumirá que esa es la **llave** para efectuar la relación.

Recuerda que las llaves primarias, siempre deben ser únicas en una base de datos. Si no lo son, las uniones generarán todas las posibles combinaciones entre los valores de la variable llave. Considera el siguiente ejemplo y observa que se repiten los valores de la variable *llave* en ambas bases


```
x ← tribble(
      ~llave, ~sexo, ~edad,
      1, "H", "25",
      2, "M", "40",
      2, "H", "18"
)

y ← tribble(
      ~llave, ~ingreso, ~gasto,
      1, "1750", "1100",
      1, "2540", "1760",
      2, "3000", "890"
)
```

```
left_join(x,y, by="llave")
```

```
## # A tibble: 4 x 5
##   llave sexo edad ingreso   gasto
##   <dbl> <chr> <chr> <chr>    <chr>
## 1 1     H     25    1750     1100
## 2 1     H     25    2540     1760
## 3 2     M     40    3000     890
## 4 2     H     18    3000     890
```

Esto generó que los datos de una misma persona por ejemplo el que se identifica con la llave 1, tenga dos posibles valores para su ingreso y su gasto. Por lo general esto es un error, ya sea en la identificación de la variables primarias o en construcción de la base.

4.4 Uniones con la ENIGH

Considera que requerimos saber cuales son los conceptos mas comunes por los cuales reciben ingreso los hombres y por cuales las mujeres. Para ello sabemos que los conceptos de ingreso se encuentran dentro de la base de datos ingresos con la variable clave, mientras que la variable que nos permite distinguir entre hombre y mujeres, se encuentra dentro de la base de datos personas. Debido a esta situación debemos buscar la forma para qué en la base de datos de ingresos, exista la variable sexo. Para ello vemos que la llave común a ambas bases se compone de *folioviv, foliohog, numren*

Como ya vimos, la función **join_left**, nos permite hacer esta **combinación de bases**.

Antes de llevar a cabo este procedimiento, vemos la base ingresos, recordemos que tiene 5 variables.

```
dim(ingresos)
```

```
## [1]    10919     5
```

```
names(ingresos)
```

```
## [1]     "folioviv" "foliohog" "numren"   "clave"    "ing_tri"
```

Ahora ejecutemos la unión o combinación de las bases

```
ingresos<- left_join(ingresos, personas, by = c("folioviv", "foliohog",
"numren"))
dim(ingresos)
```

```
## [1]     10919      11
```

```
names(ingresos)
```

```
## [1]     "folioviv" "foliohog" "numren"   "clave"    "ing_tri"
## [6]     "sexo"     "edad"     "disc1"    "hablaind" "alfabetism"
## [11]    "asis_esc"
```

Después de ejecutar **join_left**, la base contiene ahora 11 variables. Observa que a cada ingreso se le han asignado las diferentes variables contenidas en la base de personas. Observa que la nueva base tiene exactamente la misma cantidad de observaciones que la base antes de la manipulación. Esto es porque encontró una coincidencia exacta para todas las observaciones. Es decir, todos los individuos de la base de datos de ingresos, están presentes en la base de datos personas.

Hemos indicado **by = c("folioviv", "foliohog", "numren")** para especificar que es la llave presente en ambas bases de datos.

Con esta base podemos efectuar un conteo por sexo y clave para lograr nuestro propósito.

```
ingresos %>%
     count(sexo, clave) %>%
     arrange(desc(n), sexo)
```

```
## # A tibble: 132 x 3
##     sexo      clave      n
##    <chr>      <chr>      <int>
## 1  1          P001       1715
## 2  2          P001       1118
## 3  1          P009       819
## 4  2          P009       591
## 5  2          P040       524
## 6  2          P042       328
```



```
## 7  1          P022       279
## 8  1          P008       275
## 9  1          P040       270
## 10 1          P032       193
## # ... with 122 more rows
```

Las tres primeras claves del ingreso de los hombres son P001, P009 Y P022. En el caso de las mujeres son; P001, P009 Y P040. De acuerdo con el catálogo de la base estos conceptos son; P001: Sueldos y salarios; P009: Aguinaldo, P022: Ingresos por trabajos atrasados y P040: Donativos

Consideremos ahora que es de nuestro interés saber cuantas mujeres en la muestra tienen acceso a agua cada tercer día. Para esto debemos señalar que la variable **dotac_agua** de la base de datos vivienda toma los siguientes valores;

- 1 si en la vivienda hay dotación de agua diariamente
- 2 si en la vivienda hay dotación de agua cada tercer día
- 3 si en la vivienda hay dotación de agua dos veces por semana
- 4 si en la vivienda hay dotación de agua una vez por semana
- 5 si en la vivienda hay dotación de agua de vez en cuando

Sabemos que la variable que identifica a hombres y mujeres esta en la base de datos de personas, por lo cual debemos combinar la base de personas, con la información de la base de datos de viviendas.

Para ello, simplificaremos primero la base de personas

```
personas2<-select(personas, folioviv:sexo)
dim(personas2)
```

```
## [1] 7769 4
```

```
personas2<-left_join(personas2, viviendas, by="folioviv")
dim(personas2)
```

```
## [1] 7769 11
```

Pudimos haber obtenido el mismo resultado usando el operador pipe. Recuerda que el objetivo de su uso es simplificar y hacer más amigable el código. Observa que cuando lo usamos, la base sobre la cual debe operar la función sólo se escribe una vez.

```
personas2<- personas %>%
     select(folioviv:sexo) %>%
     left_join(viviendas, by="folioviv")
dim(personas2)
```

```
## [1] 7769 11
```

Si observas, ahora la base de datos de personas2 incluye la información de la vivienda. Ahora estamos listos para hacer nuestro conteo;

```
personas2 %>%
     count(sexo, dotac_agua)
```

```
## # A tibble: 12 x 3
##    sexo  dotac_agua n
##    <chr> <chr>      <int>
## 1  1     ""         169
## 2  1     "1"        3130
## 3  1     "2"        305
## 4  1     "3"        93
## 5  1     "4"        69
## 6  1     "5"        26
## 7  2     ""         163
## 8  2     "1"        3296
## 9  2     "2"        334
## 10 2     "3"        89
## 11 2     "4"        72
## 12 2     "5"        23
```

La muestra contiene 334 mujeres en cuya vivienda hay acceso a agua cada tercer día.

5 Uniones de filtro

En el ejemplo que desarrollamos anteriormente, buscamos agregar variables de la base de datos *personas* a la base de datos *ingresos*.

Las uniones de filtro relacionan observaciones de la misma forma en que lo hacen las uniones de transformación, con la diferencia que unen observaciones no variables. Hay dos tipos de uniones de filtro;

- semi_join(x,y) mantiene todas las observaciones en el objeto x, que tienen coincidencia con el objeto y.

Considera que deseamos conocer las características de las personas que trabajan. Tenemos una base de datos *trabajos* que contiene 4,205 observaciones y una base de datos de *personas* que contiene 7,769. Si hacemos;

```
personas_trabajan<-semi_join(personas, trabajos, by=c("folioviv", "foliohog",
"numren"))
```


Obtenemos una base que contiene información únicamente de las personas que trabajan. Observa que esta función conserva todas las variables de la base personas (ni una más ni una menos), pero soló las observaciones que también están presentes en trabajos. Así, podemos por ejemplo, calcular la edad promedio de las personas que trabajan.

```
mean(personas_trabajan$edad)
```

```
## [1] 38.04363
```

- anti_joint(x,y) borra todas las observaciones en el objeto x, que tienen coincidencia con el objeto y.

Por ejemplo, si quisiéramos saber la edad promedio de las personas que no trabajan, haríamos el proceso contrario, creando una base donde se identifiquen quienes no trabajan.

```
personas_no_trabajan<-anti_join(personas, trabajos, by=c("folioviv", "folio-
hog", "numren"))
```

```
mean(personas_no_trabajan$edad)
```

```
## [1] 26.23116
```

6 Problemas mas comunes con las uniones

En las bases de datos que te proporcionamos, hemos identificado con facilidad las variables de relación. Sin embargo, en la práctica cotidiana seguramente te enfrentaras con datos en los cuales es difícil identificar las variables que pueden usarse como llaves, tanto primarias como foráneas. Te damos las siguientes recomendaciones para identificarlas;

- Una variable llave, no podrá tener valores nulos, por lo cual, si crees que determina variables es una llave y observas que tiene valores perdidos, olvídala.
- Si sospechas que has identificado una variable llave foránea, asegúrate que se encuentra en alguna otra base.

Además, te recomendamos que siempre que hagas una unión observes la base resultante y analices si el procedimiento efectuado es el que buscabas.

7 Operaciones de conjuntos

Finalmente, existe un conjunto de tres operaciones que pueden efectuarse entre bases de datos. Estas funciones trabajan a nivel observaciones y las bases que se usen deben contener las mismas columnas. Estas operaciones son;

- intersect(x,y)
- union(x,y)
- setdiff(x,y)

Para ilustrarlas consideremos una base de personas y una de trabajos que tengan las mismas variables;

```
personas2<-select(personas, folioviv:numren)
trabajos2<-select(trabajos, folioviv:numren)
```

Con **intersect(x,y)** se obtienen las observaciones que están presentes tanto en la base x, como en la base y.

```
inter<-intersect(personas2, trabajos2)
```

Con **union(x,y)** se obtienen las observaciones que están presentes tanto en la base x y también las que están en la base y.

```
union<-union(personas2, trabajos2)
```

Observa que en este caso la unión es igual a la base de personas2.

Por último, **setdiff(x,y)** regresa las observaciones presentes en la base x, pero no en la base y.

```
set1<-setdiff(personas2, trabajos2)
dim(set1)
```

[1] 3941 3

Resulta que hay 3,941 observaciones, (individuos) que están en la base de personas2, pero no en la base de trabajos2

```
set2<-setdiff(trabajos2, personas2)
dim(set2)
```

```
## [1] 0 3
```

En este segundo caso, resulta que no hay ninguna observación que se encuentre en la base de trabajo2, pero no en la base de personas2.

8 Actividades

1. Determina el promedio del ingreso corriente (ing_cor) de los hogares que se ubican en viviendas que no disponen de electricidad. (aquellas donde disp_elect=5).

2. Determina cuantos hombres y cuántas mujeres viven en viviendas que tiene un estrato socioeconómico bajo (est_socio=1)

3. Determina en promedio cuanto gastan en salud los hogares cuya vivienda se ubica en vecindad. Esta característica se identifica cuando en la vivienda la variable tipo_viv toma el valor de 3

4. Determina el promedio de gasto en educación y esparcimiento (edu_espa) de los hogares, donde todos sus miembros saben leer y escribir (alfabetism=1)

5. Determina cuantas personas habitan en viviendas propias (aquellas donde la variable tenencia=4)

Capítulo 8

Cadenas de Texto

Capítulo 8 | Cadenas de Texto

1 Introducción

La manipulación de texto es una de las herramientas mas usadas para el análisis de datos en **R**. En ocasiones será necesario identificar características especificas de un texto y efectuar determinadas manipulaciones sobre ellas. En este capítulo aprenderemos los elementos básicos para la búsqueda, identificación y manipulación de patrones y coincidencias en vectores cuyos elementos son cadenas de texto. Para el desarrollo de este capítulo recurriremos a un conjunto de base de datos de palabras **words** y enunciados **sentences** que están contenidos en R. Aquí usaremos la versión original sobre el cual están construidos estos conjuntos de palabras en el idioma inglés. Si por alguna razón te sientes mas cómodo trabajando con ejemplos en el idioma español, puedes instalar la paquetería **datos**, la cual, de manera automática, proporciona una versión traducida de los conjuntos de datos. En este caso en lugar de utilizar words, puedes usar **palabras** y en lugar de sentences **oraciones**.

2 Previos

En este caso trabajaremos únicamente usando las herramientas que proporciona R por lo que no necesitaremos ninguna base de datos adicional. En este caso necesitaremos una nueva librería, la cual contienen las herramientas para el manejo de cadenas de texto. Se trata de la librería stringr, por lo cual será necesario que la instales y la actives para trabajar.

```
library(tidyverse)
library(stringr)
```


3 Creando cadenas básicas

Existen dos maneras de indicar a R que estamos trabajando con una cadena de texto. En estos momentos seguramente ya has notado que la doble comilla (") es una de ellas. La segunda es el uso de una sola comilla (')

```
cadena1<-"Esto es una cadena de texto"
cadena1
```

```
## [1] "Esto es una cadena de texto"
```

```
cadena2<-'Esto es una cadena de texto'
cadena2
```

```
## [1] "Esto es una cadena de texto"
```

El contenido delimitado entre cada inicio y cierre de comillas constituye una cadena.

Observa que en ambos casos el contenido de las variables creadas refleja la existencia de dobles comillas. Esto significa que la representación de una cadena de texto no necesariamente es la misma que la cadena en si misma. En este caso la cadena 1, parece ser diferente de la cadena 2, pues una contiene doble comillas y la otra simple. Sin embargo, su representación dentro del entorno de **R** es la misma. Esto lo podemos corroborar haciendo;

```
cadena1==cadena2
```

```
## [1] TRUE
```

Considera ahora el siguiente ejemplo;

```
a<-"\"Hola\""
a
```

```
## [1] "\"Hola\""
```

Observa que hemos incluido "\ esto significa que deseamos utilizar una doble comilla en el texto. Si deseamos conocer el contenido real de la cadena, debemos usar el comando **writeLines()**. Al usarlo nos mostrará de manera literal la palabra **"Hola"**

```
writeLines(a)
```

```
## "Hola"
```

De manera similar podemos usar \\ para indicar que en la cadena de texto deseamos que se incluya la diagonal. Es decir;

```
b<-"\\Hola"
writeLines(b)
```

```
## \Hola
```

En general el símbolo \ nos permite indicar un amplio conjunto de caracteres especiales. Por ejemplo \n indica una nueva línea, mientras que \t una tabulación.

```
c<-"Hola \n saludos"
writeLines(c)
```

```
## Hola
## saludos
```

```
d<-"Hola \t saludos"
writeLines(d)
```

```
## Hola     saludos
```

En ocasiones hay cadenas de texto escritas en patrones como **\unnnn** las cuales se encuentran en caracteres especiales usando uni-códigos (en ese caso 1–4 hex digits) los cuales tienen una representación especial. Por ejemplo

```
z<-"\u00b5"
w<-"\u0026"
z
```

```
## [1] "µ"
```

```
w
```

```
## [1] "&"
```

Puedes consultar mas códigos para caracteres especiales en esta página

Para obtener ayuda sobre el uso de " y ' puedes consultar la ayuda de R, ejecutando ?'"' en la consola.

Siempre que comiences una cadena de texto, debes indicar su cierre. En caso de no hacerlo **R** te indicará con un signo + que significa que sigue esperando indicaciones.

Anteriormente ya hemos construido múltiples cadenas de texto en un mismo vector. Para ello hacemos;

```
h<-c("Hola", "mundo", "saludos")
h
```

```
## [1] "Hola"    "mundo"    "saludos"
```

```
writeLines(h)
```

```
## Hola
## mundo
## saludos
```


Observa que en un mismo vector hay tres palabras, cada palabra esta separada por comillas y cada una constituye una cadena, por lo cual se trata de múltiples cadenas en un mismo vector.

3.1 Combinar cadenas de texto

Si es de nuestro interés combinar cadenas de texto, podemos hacer uso del comando **str_c()**.

```
x<- "Hola"
y<- "Mundo"
str_c(x, y)
```

```
## [1] "HolaMundo"
```

```
str_c(x," ", y)
```

```
## [1] "Hola Mundo"
```

```
str_c(x, y, sep=",")
```

```
## [1] "Hola,Mundo"
```

Observa que partimos de dos cadenas de texto diferentes. La cadena **x** y la cadena **y**. En el primer caso hemos creado sólo la combinación de ambas. En el segundo caso, las combinamos junto con un espacio en blanco indicado por " ". En el tercer caso las combinamos usando una , como separación entre cada cadena. De ahí que en este caso el comando se completó con la indicación **sep=","**.

Si por alguna razón en una cadena de texto existen **NA**, podemos usar el comando **str_replace_na()** para solucionarlo.

```
x <- c("abc", NA)
x
```

```
## [1] "abc" NA
```

Observa que en este caso **NA** no es reconocido como una cadena de texto, pues no incluye las comillas. Para que se reconozca como texto o podamos reemplazarlos **NA** por otra indicación hacemos;

```
str_replace_na(x)
```

```
## [1] "abc" "NA"
```

```
writeLines(str_replace_na(x))
```

```
## abc
## NA
```

```
str_replace_na(x, " ")
```

```
## [1] "abc" " "
```

```r
writeLines(str_replace_na(x, " "))
```

```
## abc
##
```

En el primer caso de manea automática se reemplaza el NA, por "NA" la cual ahora es reconocida como una cadena de texto. En el segundo caso hemos pedido que los NA se reemplacen por un espacio vacío.

Usando el carácter - podemos indicar que deseamos que a un vector que contiene cadenas de texto, se le agregue alguna expresión antes o después de cada cadena.

```r
str_c("antes-", c("x", "y", "x"), "-despues")
```

```
## [1] "antes-x-despues" "antes-y-despues" "antes-x-despues"
```

También es posible combinar cadenas de texto considerando condicionales. Por ejemplo, consideremos las cadenas de texto etiquetadas como **nombre** y **hora**, además de una variable de nombre **cumple** que únicamente guarda un valor verdadero o falso.

```r
nombre←"Arturo"
hora←"mañana"
cumple←FALSE
```

Podemos hacer la combinación de ambas cadenas de texto

```r
str_c("Buena ", hora, " ", nombre)
```

```
## [1] "Buena mañana Arturo"
```

Es posible también hacer

```r
str_c("Buena ", hora, " ", nombre,
      if (cumple) " y Feliz Cumpleaños.")
```

```
## [1] "Buena mañana Arturo"
```

La cual indica que si se obtiene un valor **TRUE** de la variable **cumple** se agregará a la cadena la expresión **"y Feliz Cumpleaños."**

```r
cumple=TRUE
str_c("Buena ", hora, " ", nombre,
      if (cumple) " y Feliz Cumpleaños.")
```

```
## [1] "Buena mañana Arturo y Feliz Cumpleaños."
```


La combinación con cadenas de texto también nos permite convertir un vector con mas de una cadena, en una sola cadena de texto. Por ejemplo;

```r
x←c("Hola", "Mundo", "Soy yo")
x
```

```
## [1] "Hola" "Mundo" "Soy yo"
```

Dentro del vector **x** hay tres cadenas de texto (recuerda que cada cadena se identifica por el inicio y cierre de comillas).

Podemos transformar **x** en una sola cadena de texto. Para ello usamos la indicación de **collapse** dentro del comando.

```r
str_c(x, collapse = " ")
```

```
## [1] "Hola Mundo Soy yo"
```

Ahora tenemos una sola cadena de texto. Observa que hemos indicado que se efectúe un collapse usando el espacio en blanco. Podemos también hacer collapse = ".", o collapse = ","

```r
str_c(x, collapse = ".")
```

```
## [1] "Hola.Mundo.Soy yo"
```

```r
str_c(x, collapse = ",")
```

```
## [1] "Hola,Mundo,Soy yo"
```

```r
str_c(x, collapse = "_")
```

```
## [1] "Hola_Mundo_Soy yo"
```

3.2 Subconjuntos de cadenas

Así como es posible combinar cadenas de texto, también podemos obtener subconjuntos de una cadena. En capítulos anteriores hemos trabajo un poco esta idea. Considera que hay un vector con varias cadenas de texto y que por alguna razón necesitamos extraer los tres primeros caracteres de cada cadena. Para ello usamos el comando str_sub()

```
x ← c("0014525", "0024685", "0028596")
str_sub(x, 1, 3)
```

```
## [1] "001" "002" "002"
```

La indicación 1,3 refiere a que deseamos obtener desde el carácter 1 hasta el 3.

Si deseamos obtener los últimos cuatro caracteres de cada cadena, hacemos;

```
str_sub(x, -4, -1)
```

```
## [1] "4525" "4685" "8596"
```

Podemos combinar str_sub con otros comandos. Por ejemplo

```
y←c("Manzana", "Plátano", "Pera")
str_to_lower(str_sub(y, 1, 4))
```

```
## [1] "manz" "plát" "pera"
```

Por lo cual, el comando str_to_lower() cambia mayúsculas a minúsculas.

También es posible usar los comandos **str_to_upper(), str_to_title()**.

```
str_to_upper("hola")
```

```
## [1] "HOLA"
```

```
str_to_title("hola que tal")
```

```
## [1] "Hola Que Tal"
```

Otra de las funciones cuando se trabaja con cadenas de texto, es poder ordenarlas. Consideremos el siguiente vector que contiene cinco cadenas de texto

```
x←c("Anual", "Trimestral", "Semestral", "Uso", "Equipo")
```

Para ordenar las cadenas usamos **str_sort()** y **str_order()**

```
str_sort(x)
```



```
## [1] "Anual"   "Equipo"   "Semestral"     "Trimestral" "Uso"
```

```
str_order(x)
```

```
## [1] 1 5 3 2 4
```

La primera instrucción regresa las cadenas de texto en ordenadas alfabéticamente. La segunda regresa el número que le correspondería a cada cadena en el ordenamiento.

3.3 Configuraciones locales

Algunos de los comandos que hemos usado requieren de una configuración específica. Por ejemplo;

```
str_to_upper(c("i", "ı"))
```

```
## [1] "I" "I"
```

```
str_to_upper(c("i", "ı"), locale = "tr")
```

```
## [1] "I" "I"
```

Esto es porque dentro del Turco existen dos **i**, una con punto y otra sin el. Al indicar específicamente que deseamos se considere ese idioma, hace la conversión a mayúsculas bajo ese supuesto. Si no le indicamos la configuración local, entiende que debe usar la predeterminada que siempre será en el idioma *Inglés*.

Una mas de las configuraciones locales, es el uso de los comandos **str_sort()** y **str_order().** Por lo general en el español y el inglés el ordenamiento es según el orden alfabético. En el caso de idiomas como el hawaiano, el ordenamiento indica que van primero las vocales que las consonantes. En este caso tendríamos;

```
x←c("Anual", "Trimestral", "Semestral", "Uso", "Equipo")
str_sort(x, locale = "haw")
```

```
## [1] "Anual" "Equipo" "Uso" "Semestral" "Trimestral"
```

```
str_order(x,locale = "haw")
```

```
## [1] 1 5 4 3 2
```

4 Identificando patrones

Cuando se tiene una gran cantidad de cadenas de texto, suele ser necesario identificar determinados patrones en ellas. Para ello es necesario el uso de comando como **str_view()** y **str_view_all().** Comencemos identificando algunos patrones básicos.

4.1 Patrones básicos

Consideremos el siguiente vector que tiene varias cadenas de texto.

```
x ← c("arroz", "arma", "azúcar", "arte", "turca", "martes", "marzo")
```

Supongamos que del vector **x** nos interesa identificar aquellas cadenas de texto que contengan el patrón *ar.* Para ello hacemos

```
str_view(x, "ar")
```

```
arroz
arma
azucar
arte
turca
martes
marzo
```

Observemos que como resultado se nos muestran todas las cadenas de texto, resaltando aquellas en las que se identifica el patrón deseado, no importando en que parte de la cadena se encuentre.

Si en lugar de eso, deseamos identificar aquellas cadenas en las que se incluye la letra **a** y a la derecha se encuentre cualquier otra letra, tenemos que usar el punto .. Es decir;

```
str_view(x, "a.")
```

```
arroz
arma
azucar
arte
turca
martes
marzo
```


Observa que la palabra turca no se identifica pues aunque contiene la letra **a,** a su derecha no hay mas elementos.

Ahora, si buscamos identificar en las cadenas de texto, aquellas donde se encuentra la letra **a** seguida y antecedida por cualquier otra letra, debemos poner el punto a ambos lados de la letra **a.** Esto es;

```
str_view(x, ".a.")
```

```
arroz
arma
azucar
arte
turca
martes
marzo
```

Como podrás observar el punto, nos permite indicar que el patrón que buscamos requiere cualquier carácter que acompañe a determinada letra.

En caso de que en el patrón a identificar deseemos ubicar el punto debemos incluir \\. Igualmente, si deseamos ubicar el símbolo \ en un patrón de texto, debemos incluir \\\\

Por ejemplo, con la indicación siguiente estamos solicitando se identifique en la cadena de texto, el punto .

```
h←c("xyz", "x.y", "w.p\\q", "py.7", "dd.q", "df\\ded", "qrt", "a\\b")
str_view(h, "\\.")
```

```
xyz
x.y
w.p\q
py.
dd.q
df\ded
qrt
a\b
```

En el siguiente ejemplo solicitamos se identifique la diagonal \

```
str_view(h, "\\\\")
```

```
xyz
x.y
w.p\q
py.
dd.q
df\ded
qrt
a\b
```

4.2 Anclas

Hasta ahora hemos buscado patrones determinados en cualquier parte de una cadena, pudiendo ser al principio, al final o en medio de ella. Si es de nuestro interés considerar únicamente la coincidencia al inicio de la cadena debemos usar ^. En cambio, si deseamos buscar coincidencia al final usamos **$**.

Al inicio de la cadena;

```
x ← c("arroz", "arma", "azucar", "arte", "turca", "martes", "marzo")
str_view(x, "^a")
```

```
arroz
arma
azucar
arte
turca
martes
marzo
```


Al final de la cadena

```
str_view(x, "a$")
```

```
arroz
arma
azucar
arte
turca
martes
marzo
```

Para forzar a que una expresión coincida con una cadena de texto, debemos anclarla tanto al inicio como al final.

```
y←c("pastel de manzana", "manzana", "jugo de manzana", "pure de manzana")
str_view(y, "manzana")
```

```
pastel de manzana
manzana
jugo de manzana
pure de manzana
```

```
str_view(y, "^manzana$")
```

```
pastel de manzana
manzana
jugo de manzana
pure de manzana
```

Observa que en el primer caso cuando solo indicamos manzana se identifica tanto al inicio como al final de la cadena. En cambio en el segundo caso, cuando señalamos la palabra con anclas, se identifica únicamente la cadena que contiene exactamente la palabra buscada.

4.3 Otros caracteres

Así como el punto . nos permite identificar coincidencias con cualquier carácter, podemos identificar cuatro herramientas mas que son de utilidad.

- \d: busca coincidencia con cualquier dígito
- \s: coincidencias con espacios en blanco
- [abc]: coincide con a,b o c
- [ˆabc]: coincide con todo menos a,b o c

En el caso de \d, \s, deberás usar la doble barra para indicar su búsqueda.

En este primer ejemplo indicamos que deseamos que se busque aquellas cadenas que contengan a[.*]c (a seguida de punto o asterisco y luego seguido de c)

```
h<-c("a3bc", "a.c", "a*c", "a c", "abc", "a5", "b*c")
str_view(h, "a[.*]c")
```

```
a3bc
a.c
a*c
a c
abc
a5
b*c
```

En este caso el patrón buscado *a[ˆ.]c* (a seguida de cualqueir cosa excepto punto y luego c)

```
str_view(h, "a[^.]c")
```

```
a3bc
a.c
a*c
a c
abc
a5
b*c
```


Letra a seguida de cualquier dígito

```
str_view(h, "a\\d")
```

```
a3bc
a.c
a*c
a c
abc
a5
b*c
```

Letra a seguida de espacio en blanco

```
str_view(h, "a\\s")
```

```
a3bc
a.c
a*c
a c
abc
a5
b*c
```

En este caso el patrón .[*]c es cualquier cosa, seguida de asterisco seguida de c.

```
str_view(h, ".[*]c")
```

```
a3bc
a.c
a*c
a c
abc
a5
b*c
```

En algunas ocasiones será necesario usar \\ para identificar ciertos caracteres. Por ejemplo para buscar -

```
w<-c("H-q", "H^q", "H]q")
str_view(w, "\\-")
```

```
h-q
h^q
h]q
```

Para buscar ^

```
str_view(w, "\\^")
```

```
h-q
h^q
h]q
```

Para buscar]

```
str_view(w, "\\]")
```

```
h-q
h^q
h]q
```

Es posible también identificar patrones alternativos usando | correspondiente a la o lógica. El siguiente ejemplo muestra la petición de identificar un patrón que incluya en la cadena] o ^. Observa que se han usado paréntesis al igual que se haría con cualquier expresión lógica o matemática.

```
w<-c("H-q", "H^q", "H]q")
str_view(w, "(\\])|(\\^)")
```

```
h-q
h^q
h]q
```


4.4 Repetición

Hasta ahora, lo patrones que hemos buscado se encuentran únicamente una vez en la cadena de texto. Mediante el uso de ?,+ podemos controlar el número de veces que queremos que se encuentre el patrón que buscamos.

- ? Señalará las cadenas donde el patrón se repita 0 o una vez.
- + Señalará las cadenas donde el patrón se repita 1 o más veces.

También es posible identificar el número de coincidencias de manera precisa. Para ello hacemos uso de;

- {n}: identifica el patrón exactamente n veces
- {n,}: identifica el patrón n o más veces
- {,m}: identifica el patrón mas de m veces
- {n,m}: identifica el patrón entre n y m veces

Identifica el patrón CC una vez

```
h<- "MDCCCCCCCCCCLXXXVIII"
str_view(h, "CC?")
```

```
MDCCCCCCCCCCLXXXVIII
```

Identifica al patrón *CC* más de una vez. En este caso se repite cinco veces

```
str_view(h, "CC+")
```

```
MDCCCCCCCCCCLXXXVIII
```

Indetifica el patrón *C* cuatro veces

```
str_view(h, "C{4}")
```

```
MDCCCCCCCCCCLXXXVIII
```

Identificar el patrón *C* cuatro o mas veces

```
str_view(h, "C{4,}")
```

```
MDCCCCCCCCCCLXXXVIII
```

Identificar el patrón *C* entre 4 y 6 veces

```
str_view(h, "C{4,6}")
```

```
MDCCCCCCCCCCLXXXVIII
```

En el primer caso el resultados muestra que se cumple el patrón buscado en todas las cadenas, pues hemos indicado ?. En el segundo caso, hemos indicado +, por lo cual se consideran solo las cadenas donde el patrón completo aparece una o más veces.

4.5 Agrupamiento

En ocasiones deseamos identificar en cadenas de texto que coinciden con una parte regular de la cadena. Por ejemplo la palabra papaya contiene un par de letras repetido, lo mismo que banana y coco. Podemos buscar estos patrones haciendo;

```
frutas<-c("banana", "papaya", "lima", "coco", "naranja")
str_view(frutas, "(..)\\1", match = TRUE)
```

```
banana
papaya
coco
```

Donde hemos indicado que se identifiquen aquellas cadenas que tengan un par de letras repetido (mediante la indicación \\1).

5 Herramientas

Hasta ahora hemos trabajado con algunas expresiones básicas para identificar patrones en una cadena de texto y hemos visto el proceso mediante el cual se identifican estos patrones. Ampliaremos los conocimientos en este apartado usando algunas otras funciones de utilidad que se encuentran en la paquetería de **stringr()**.

5.1 Detectar coincidencias y encontrar su posición

- Para determinar qué cadenas coinciden con un patrón usaremos la función **str_detect()**

```
frutas<-c("banana", "papaya", "lima", "coco", "naranja", "pera")
# Cadenas que contienen la letra e
str_detect(frutas, "e")
## [1] FALSE FALSE FALSE FALSE FALSE TRUE
```

```
# Cadenas que contienen la letra a
str_detect(frutas, "a")
## [1] TRUE TRUE TRUE FALSE TRUE TRUE
```



```
# Cadenas que empiezan con p
str_detect(frutas, "^p")
## [1] FALSE TRUE FALSE FALSE FALSE TRUE
```

```
# Cadenas que terminan con vocal
str_detect(frutas, "[aáeéiéoéuú]$")
## [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

```
# Contar las cadenas que empiezan con p
sum(str_detect(frutas, "^p"))
## [1] 2
```

Observa que en este caso los resultados consisten en una conjunto de valores FALSE TRUE indicando si la cadena de texto en determinada posición cumple con la coincidencia buscada.

Por ejemplo, en el primero caso **str_detect(frutas, "e")** el resultado arroja **FALSE FALSE FALSE FALSE FALSE TRUE** lo que indica que solo la cadena que se encuentra en la entrada 6, cumple con la coincidencia de contener la letra e.

Una forma alternativa de contar cuantas cadenas en el vector cumplen con la coincidencia es usar **str_count()**, la cual nos indica el numero de veces que la coincidencia esta presente en la cadena.

```
str_count(frutas, "p")
## [1] 0 2 0 0 0 1
```

En este caso tenemos que la coincidencia p se encuentra dos en la cadena 2 (“papaya”) y una vez en la cadena 6 (“pera”).

De ser necesario buscar condiciones lógicas con mayor grado de complejidad, podemos utilizar la función **str_detect()** mas de una vez.

Para ilustrar el siguiente ejemplo, utilizaremos el conjunto **words**, por medio del cual **R** de manera automática identifica un conjunto de 980 palabras del idioma inglés. Recuerda que si te sientes mas cómodo trabajando con palabras en español, puedes instalar la paquetería *datos* que mencionamos al inicio de este capítulo y usar las bases de datos de nombre *palabras* en lugar de *words* y *oraciones* en lugar de *sentences*.

Las primeras 10 palabras contenidas en words se muestran como ejemplo:

```
length(words)
## [1] 980
```

```
words[1:10]
## [1] "a"       "able"     "about"    "absolute" "accept"   "account"
## [7] "achieve"      "across"   "act"            "active"
```

De este conjunto de palabras buscaremos todas aquellas que no contengan ninguna vocal. Para detectar cuales palabras tienen esta coincidencia, primero buscaremos todas las palabras que tengan al menos una vocal. Luego buscaremos el contrario de este conjunto.

```
una_vocal←str_detect(words, "[aáeéiíoóuú]")
sin_vocal←!una_vocal
```

Observa que hemos usado el símbolo **!** indicando que es diferente.

Si exploras el contenido dentro de **una_vocal, sin_vocal** te darás cuenta qué únicamente contienen valores falsos o verdaderos, tal y como lo mencionamos anteriormente.

Si deseamos ver cuales son las palabras que cumplen la coincidencia hacemos;

```
words[sin_vocal]
## [1] "by" "dry" "fly" "mrs" "try" "why"
```

Es posible hacer un subconjunto de **words** que contenga únicamente las cadenas deseadas, para ello usamos la función **str_subset()**. Por ejemplo, si deseamos las cadenas que empiecen con *l*;

```
str_subset(words, "^l")
## [1] "labour"  "lad"      "lady"     "land"     "language" "large"
## [7] "last"    "late"     "laugh"    "law"      "lay"      "lead"
## [13] "learn"  "leave"    "left"     "leg"      "less"     "let"
## [19] "letter" "level"    "lie"      "life"     "light"    "like"
## [25] "likely" "limit"    "line"     "link"     "list"     "listen"
## [31] "little" "live"     "load"     "local"    "lock"     "london"
## [37] "long"   "look"     "lord"     "lose"     "lot"      "love"
## [43] "low"    "luck"     "lunch"
```


5.2 Encontrar la posición

Consideremos ahora que el interés no es conocer cuales son las cadenas que cumplen con una coincidencia, sino mas bien, saber cual es su posición (en términos numéricos) dentro del vector de cadenas de texto. Para ello pondremos el conjunto de cadenas de texto contenidas en el vector **words** en un *data frame* y asignaremos un número a cada cadena.

```
df ← tibble(
     word = words,
     i = seq_along(word)
     )
df
## # A tibble: 980 x 2
##    word       i
##    <chr>      <int>
## 1  a          1
## 2  able       2
## 3  about      3
## 4  absolute   4
## 5  accept     5
## 6  account    6
## 7  achieve    7
## 8  across     8
## 9   act       9
## 10 active     10
## # ... with 970 more rows
```

Observa que la función **seq_along()** únicamente agrega un valor numérico consecutivo a cada cadena contenida en el vector.

Ahora podemos usar un filtro para identificar que posición dentro del conjunto de 980 cadenas, ocupan las cadenas que empiezan con **l**.

```
df %>%
filter(str_detect(words, "^l"))
## # A tibble: 45 x 2
##    word       i
## <  chr>       <int>
## 1  labour     452
## 2  lad        453
```

```
## 3  lady        454
## 4  land        455
## 5  language    456
## 6  large       457
## 7  last        458
## 8  late        459
## 9  laugh       460
## 10 law         461
```

... with 35 more rows

Con estos elementos podemos contabilizar cuantas vocales y cuantas consonantes existen en cada cadena del vector **words**

```
df %>%
     mutate(
          vocales=str_count(word, "[aeiou]"),
          consonantes=str_count(word, "[^aeiou]")
     )
```

```
## # A tibble: 980 x 4
##    word       i          vocales    consonantes
##    <chr>      <int>      <int>      <int>
## 1  a          1          1          0
## 2  able       2          2          2
## 3  about      3          3          2
## 4  absolute   4          4          4
## 5  accept     5          2          4
## 6  account    6          3          4
## 7  achieve    7          4          3
## 8  across     8          2          4
## 9  act        9          1          2
## 10 active     10         3          3
## # ... with 970 more rows
```

En este ejemplo hemos usado **[aeiou]** recuerda que esta opción identifica cualquier los contenidos *a,e,i,o,u* y la función **str_count** las contabiliza. Por otro lado, **[^aeiou]** identifica los contenidos que no sean, ni *a,e,i,o,u*.

- Extraer el contenido de las coincidencias

Una vez que identificamos las coincidencias en una cadena de texto y podemos ubicar la posición en que se encuentran, podemos extraer los contenidos. En este caso para ejemplificar el proceso, usaremos el vector **sentences** que incluye un conjunto de 720 enunciados en inglés. Se muestran los primeros cinco enunciados.

```
length(sentences)
```

```
## [1] 720
```

```
sentences[1:5]
```

```
## [1] "The birch canoe slid on the smooth planks."
## [2] "Glue the sheet to the dark blue background."
## [3] "It's easy to tell the depth of a well."
## [4] "These days a chicken leg is a rare dish."
## [5] "Rice is often served in round bowls."
```

Consideremos que de este conjunto deseamos identificar aquellos enunciados que contiene alguna de las siguientes palabras; dark, dish, chicken, will, like.

Para ello primero construiremos un vector que contenga las palabras a buscar en la cadena de texto.

```
coincidencia<-"dark|dish|chicken|will|like"
```

Observa que hemos incluido todas las palabras como una sola cadena, separando por medio del carácter | el cual representa la o lógica. Por lo que busqueda se enfocara en las cadenas que contengan *dark* o *dish* o *chicken*, etc.

Ahora crearemos un subconjunto con los enunciados que contienen estas palabras. Para ello usamos **str_subset()**. Vemos que hay 42 enunciados con la coincidencia. Se muestran solo los primeros cinco

```
tiene_coincidencia<-str_subset(sentences, coincidencia)
length(tiene_coincidencia)
```

```
## [1] 42
```

```
tiene_coincidencia[1:5]
```

```
## [1] "Glue the sheet to the dark blue background."
## [2] "These days a chicken leg is a rare dish."
## [3] "The rope will bind the seven books at once."
## [4] "The bark of the pine tree was shiny and dark."
## [5] "A rag will soak up spilled water."
```

Ahora vamos a extraer de cada enunciado, la coincidencia detectada. Para ello usaremos la función **str_extract()**. Se muestran nuevamente las primeras cinco coincidencias detectadas.

```
coin_encontrada<-str_extract(tiene_coincidencia, coincidencia)
length(coin_encontrada)
```

```
## [1] 42
```

```
coin_encontrada[1:5]
```

```
## [1] "dark" "chicken" "will" "dark" "will"
```

Con esto, hemos extraído las coincidencia detectadas en cada cadena. Observa que la segunda cadena de texto *These days a chicken leg is a rare dish*, contiene dos coincidencias (chicken y dish). Sin embargo, al ejecutar la instrucción anterior, el resultado reporta sólo una coincidencia con la palabra *chicken*. Esto es porque se trata de la primera coincidencia y la función se enfoca sólo en la primera detectada. Para ampliar la identificación y considerar los casos en los que la cadena tenga mas de una coincidencia hacemos:

```
mas_coin<-sentences[str_count(sentences, coincidencia)>1]
```

Observa que hemos indicado que en el conteo se identifique mas de una coincidencia. De esta manera dentro del vector **mas_coin** estarán contenidos únicamente los enunciados donde existe más de una coincidencia.

Podemos comprobarlo haciendo

```
str_view_all(mas_coin, coincidencia)
```

Encontramos que hay una sola cadena que contiene mas de una coincidencia. Para extraerla usamos la función **str_extract_all()**

```
str_extract(mas_coin, coincidencia)
```

```
## [1] "chicken"
```

```
str_extract_all(mas_coin, coincidencia)
```

```
## [[1]]
## [1] "chicken" "dish"
```


Observa que usar la función **str_extract()** falla, pues solo detecta la primera coincidencia, mientras que **str_extract_all()** detecta ambas. Si en esta función agregamos **simplify = TRUE** obtendremos una matriz donde se muestran las coincidencias separadas en columnas.

```
str_extract_all(mas_coin, coincidencia, simplify = TRUE)
```

```
## [,1] [,2]
## [1,] "chicken" "dish"
```

5.3 Remplazar coincidencias

En ocasiones se requiere que una vez identifica una determina coincidencia, esta sea reemplazada. Para ello usamos **str_replace() str_replace_all()** la primera de estas funciones reemplazara únicamente la primera coincidencia, mientras que la segunda las reemplazará todas.

```
frutas<-c("banana", "papaya", "lima", "coco", "naranja", "pera")
str_replace(frutas, "[aeiou]", "-")
```

```
## [1] "b-nana"  "p-paya"   "l-ma"     "c-co"     "n-ranja"  "p-ra"
```

```
str_replace_all(frutas, "[aeiou]", "-")
```

```
## [1] "b-n-n-"  "p-p-y-"   "l-m-"     "c-c-"     "n-r-nj-"  "p-r-"
```

5.4 Dividir una cadena

En ocasiones será necesario que el contenido de una cadena sea divido en varias cadenas. Consideremos primero que deseamos separar el contenido de un enunciado en palabras. Para ejemplificarlo construiremos un vector de enunciados con únicamente las primeras 5 oraciones del vector **sentences.** Para separarlas usaremos la función **str_split()**

```
enunciados<- sentences[1:5] %>%
     str_split(" ", simplify = TRUE)
enunciados
```

```
##         [,1]  [,2]     [,3]  [,4]    [,5]  [,6]        [,7]     [,8]
## [1,]    "The" "birch" "canoe" "slid"  "on"  "the"    "smooth"   "planks."
## [2,]    "Glue" "the"  "sheet" "to"    "the" "dark"   "blue"  "background."
## [3,]    "It's" "easy"    "to" "tell"  "the" "depth"  "of"           "a"
## [4,]    "These" "days"    "a" "chicken" "leg"  "is"  "a"        "rare"
## [5,]    "Rice" "is"   "often" "served" "in" "round" "bowls."        ""
## [,9]
```

```
## [1,]     ""
## [2,]     ""
## [3,]     "well."
## [4,]     "dish."
## [5,]     ""
```

Observa que se ha obtenido una separación en palabras, gracias a que hemos indicado que la separación será cada espacio vacío " ". Podemos separar también por medio de otros caracteres, por ejemplo :

```
registro←c("Nombre:Arturo","Pais:Mex", "Edad:28") %>%
     str_split(":", simplify = TRUE)
registro
```

```
##          [,1]       [,2]
## [1,]     "Nombre"   "Arturo"
## [2,]     "Pais"     "Mex"
## [3,]     "Edad"     "28"
```

5.5 Otros patrones

Cuando se identifica un patrón, entra de manera automática en función la opción para identificar una expresión regular, **regex**. Observa que en el siguiente ejemplo, obtenemos el mismo resultado de dos maneras diferentes:

```
bananas ← c("banana", "Banana", "BANANA")
str_view(bananas, "banana")
str_view(bananas, regex("banana"))
```

```
banana
Banana
BANANA
```

En caso de que se añadan opciones a la función **str_view()**, es necesario incluir la opción **regex()**. Las opciones mas comunes de esta función son;

- ignore_case = TRUE desestima el uso de mayúsculas y minúsculas
- multiline = TRUE permite identificar patrones al inicio y al final de cada linea, en lugar de que sea en toda la cadena
- comments = TRUE permite el uso de comentarios en la instrucción

Por ejemplo, cuando no deseas que se diferencien entre mayúsculas y minúsculas en una identificación de un patrón en una cadena, se debe especificar **ignore_case = TRUE**. En este caso hay que indicar que se trata de una expresión regular usando

```
bananas ← c("banana", "Banana", "BANANA")
str_view(bananas, regex("banana", ignore_case = TRUE))
```

```
banana
Banana
BANANA
```

Observa que hemos ignorado las mayúsculas y se identifica el patrón independiente de ellas.

Ahora, para identificar patrones en diferentes lineas de una cadena, consideremos;

```
x ← "Linea1\nLinea2\nLinea3"
writeLines(x)
```

```
## Linea1
## Linea2
## Linea3
```

Al usar **writeLines(x)** nos damos cuenta que se trata de una cadena con tres lineas. Al efectuar la búsqueda como lo hemos hecho anteriormente, observamos que se identifica sólo una vez

```
str_extract_all(x, "^Linea")[[1]]
```

```
## [1] "Linea"
```

En cambio cuando agregamos **multiline = TRUE** se buscará la coincidencia en todas las líneas de la cadena.

```
str_extract_all(x, regex("^Linea", multiline = TRUE))[[1]]
```

```
## [1] "Linea" "Linea" "Linea"
```

R posee una paquetería que no requiere ninguna instalación extra y que es propia del código donde **R** ha sido construido. Esta paquetería lleva por nombre **stringi()** y contiene funciones similares a las que hemos señalado sobre **stringir()**. Sin embargo, las funciones de stringir() resultan ser mas sencillas y mas intuitivas, por lo que nos hemos enfocado en el uso de esta paquetería para entender los procesos básicos del manejo de cadenas de texto en **R.**

6 Actividades

1. Usa el vector de cadenas de texto palabras para determinar lo siguiente;

- Un subconjunto donde se encuentren las palabras que comiencen con h
- Encuentra las palabras que tienen 5 o más vocales
- Encuentra las palabras que se ubican en la posición 560 y 625
- Toma las primeras 5 palabras (cadenas) y agrúpalas todas en una sola cadena de texto usando (/)

2. Usa el vector de cadenas de texto oraciones para determinar lo siguiente;

- Todas las cadenas que contienen almenos una vez una de las siguientes coincidencias; rojo, azul, amarillo, verde.
- Usa el resultado anterior para identificar cuantas veces existe la coincidencia en la cadena.
- Construye un data frame que en una columna tenga los enunciados que presentan coincidencia y en la siguiente el numero de coincidencias encontradas

3. Usa el conjunto de cadenas de texto de nombre oraciones para determinar cuales son las cadenas que contienen las siguientes características;

- Algún acento. Construye un subconjunto con tales cadenas
- El número de a,e,i,o, u en cada cadena, para determinar cual vocal es la mas utilizada
- Las cadenas que empiecen con una consonante y terminen con vocal

Capítulo 9

Manejo de fechas

Capítulo 9 | Manejo de fechas

1 Introducción

En este capítulo trabajaremos los elementos para el manejo de fechas y fechas con tiempos. Aprenderemos también a configurar una zona horaria determinada en **R**. Veremos el efecto que tienen algunas operaciones aritméticas con las variables de tipo fecha y tiempo. Para el desarrollo de las actividades de este capítulo, será necesario el uso de la base de datos de nombre ´mibici´.

2 Previos

Para trabajar con fechas es necesario agregar una nueva librería a nuestro entorno de trabajo. Esta librería lleva por nombre lubridate() por lo cual lo primero que necesitamos es instalarla y cargarla al entorno de R. Necesitamos ademas la librería que contiene la funciones básicas con las que hemos trabajo previamente tidverse() y readr() para cargar la base de datos con la que trabajaremos.

```
library(tidyverse)
library(readr)
library(lubridate)
```

3 Creando fechas y tiempos

En **R** podemos encontrar tres tipos diferentes de datos que se relacionan con fechas y tiempo:

- Fechas: En este caso se refiere únicamente a una fecha, por ejemplo 01 de enero de 2020. Las variables de fecha se reconocen en **R** como **<date>**.

- Tiempo: Este caso se refiere únicamente una hora, minuto y segundo específico. Las variables de tiempo se reconocen en **R** como **<time>**.
- Fechas y tiempo: Una variable de tipo fecha y tiempo, contiene ambas cosas. Una variable de este tipo se identifica en **R** como **<dttm>**. En ocasiones pueden ser identificada también como *POSIXct*.

Como siempre, es recomendable mantener las cosas lo mas sencillo posible, por lo cual, sí la esencia de una variable es la fecha, lo ideal es mantenerla como fecha, a menos que el tiempo sea importante y deba rescatarse.

Empecemos por lo básico. Probemos las funciones **today(), now()**. La primera de ellas nos regresa la fecha, actual. La segunda nos regresa una fecha y una hora que responde a un preciso instante.

```
today()
```

```
## [1] "2021-01-09"
```

```
now()
```

```
## [1] "2021-01-09 14:38:56 CST"
```

Si indagamos el tipo de entrada o de dato, que es cada una de ellas, hacemos;

```
class(today())
```

```
## [1] "Date"
```

```
class(now())
```

```
## [1] "POSIXct" "POSIXt"
```

Observa que el resultado esta dado en términos de año, mes y día. La hora que se muestre dependerá de la zona horaria, en la que se encuentre la configuración de **R**. En el apartado final de este capítulo aprenderemos a identificar y cambiar zonas horarias.

Con el uso de la librería **ludridate()**, disponemos de tres formas distintas de crear una fecha/tiempo.

- Desde una cadena de texto
- Desde componentes individuales de fecha y tiempo
- Desde un objeto que contiene fechas y tiempo

3.1 Desde una cadena de texto

Considera que tenemos una cadena de texto que en realidad representa una fecha. Por ejemplo, en seguida tenemos tres formas diferentes de escribir una misma fecha. Observa que la fecha se encuentra en el formato del idioma inglés, pues es la construcción original de **R**.

```
fecha1<-"2021-01-05"
fecha2<-"January 05, 2021"
fecha3<-"05-Jan-2021"
class(fecha1)
```

```
## [1] "character"
```

```
class(fecha2)
```

```
## [1] "character"
```

```
class(fecha3)
```

```
## [1] "character"
```

Si solicitamos el tipo de objeto de que se trata, vemos que es una cadena de texto.

Podemos convertir cada una de estas cadenas de texto a un formato que sea reconocido como una fecha. Para ello podemos hacer lo siguiente;

```
fecha1<-ymd(fecha1) #El formato de la cadena es; año, mes, día
fecha2<-mdy(fecha2) #El formato de la cadena es; mes, día, año
fecha3<-dmy(fecha3) ##El formato de la cadena es; día, mes, año
class(fecha1)
```

```
## [1] "Date"
```

```
class(fecha2)
```

```
## [1] "Date"
```

```
class(fecha3)
```

```
## [1] "Date"
```

```
fecha1
```

```
## [1] "2021-01-05"
```



```
fecha2
```

```
## [1] "2021-01-05"
```

```
fecha3
```

```
## [1] "2021-01-05"
```

Observa que en los tres casos obtenemos ahora un objeto de tipo fecha y como resultado se establece el mismo formato de fecha. Observa también que hemos usado tres funciones distintas dependiendo de la forma en la que se encuentra la representación de la fecha en la cadena de texto.

En el caso del primero formato en la fecha1 se puede efectuar la conversión aún si carecemos de las comillas.

```
fecha1<-20210105
fecha1<-ymd(fecha1)
fecha1
```

```
## [1] "2021-01-05"
```

```
class(fecha1)
```

```
## [1] "Date"
```

Para crear fechas que incluyan horas, usaremos las funciones **ymd_hmd()** y **mdy_hm().**

```
hora1<-"2021-01-05 11:53:23"
hora2<-"05/01/2021 08:01"
hora1<-ymd_hms(hora1)
hora2<-mdy_hm(hora2)
hora1
```

```
## [1] "2021-01-05 11:53:23 UTC"
```

```
hora2
```

```
## [1] "2021-05-01 08:01:00 UTC"
```

Observa que para la **hora2** usamos **mdy_hm** ya que el formato no incluye segundo. Existen otras funciones que se ejemplifican a continuación;

```
mdy_h("05/01/2021 08")
```

```
## [1] "2021-05-01 08:00:00 UTC"
```

```
mdy_hm("05/01/2021 08:01")
## [1] "2021-05-01 08:01:00 UTC"

mdy_hms("05/01/2021 08:01:34")
## [1] "2021-05-01 08:01:34 UTC"

ymd_h("2021-01-05 22")
## [1] "2021-01-05 22:00:00 UTC"

ymd_hm("2021-01-05 22:53")
## [1] "2021-01-05 22:53:00 UTC"

ymd_hms("2021-01-05 22:53:23")
## [1] "2021-01-05 22:53:23 UTC"

dmy_h("05-Jan-2021 09")
## [1] "2021-01-05 09:00:00 UTC"

dmy_hm("05-Jan-2021 09:43")
## [1] "2021-01-05 09:43:00 UTC"

dmy_hms("05-Jan-2021 09:43:20")
## [1] "2021-01-05 09:43:20 UTC"
```

Observa en que casos conviene usar que función para transformar el texto en fecha.

3.2 Desde componentes individuales

Considera ahora que tenemos una tabla con información (base de datos), que contiene diversas columnas y en cada columna hay un componente de la fecha. Para ello usaremos la base de datos de viajes del programa *Mi Bici* un programa de transporte en la Ciudad de Guadalajara. Esta base de datos contiene información sobre el inicio y el final del viaje en bicicleta que realizó cada usuario durante el mes de diciembre del 2020.


```
mibici<-read_csv("mibici.csv")
##
## -- Column specification ------------------------------------------------
-------
##   cols(
##   viaje_id = col_double(),
##   usuario_id = col_double(),
##   genero = col_character(),
##   nacimiento = col_double(),
##   origen_id = col_double(),
##   destino_id = col_double(),
##   year = col_double(),
##   month = col_double(),
##   day = col_double(),
##   hour_inicio = col_double(),
##   minute_inicio = col_double(),
##   second_inicio = col_double(),
##   hour_final = col_double(),
##   minute_final = col_double(),
##   second_final = col_double()
## )

mibici
## # A tibble: 21,316 x 15
## viaje_id usuario_id genero nacimiento origen_id destino_id year month day
##   <dbl>      <dbl>      <chr> <dbl>      <dbl> <dbl>      <dbl><dbl> <dbl>
## 1 17969550   29296      M     1949       37    260        2020  12    22
## 2 17975837   3878       M     1965       9     36         2020  12    23
## 3 17904109   28966      M     1982       236   226        2020  12    15
## 4 17814561   80048      F     1998       248   35         2020  12    6
## 5 17838913   77001      M     1989       272   83         2020  12    9
## 6 17900196   78651      F     1991       272   182        2020  12    15
## 7 17945045   73653      F     1991       260   2          2020  12    19
## 8 17953328   76055      M     1989       67    194        2020  12    21
## 9 17893294   71308      F     1993       26    24         2020  12    14
```

```
## 10 17810347   74085       M     1977       2     31          2020  12    5
## # ... with 21,306 more rows, and 6 more variables: hour_inicio <dbl>,
## # minute_inicio <dbl>, second_inicio <dbl>, hour_final <dbl>,
## # minute_final <dbl>, second_final <dbl>
```

Observa que la tabla de datos contiene información sobre el inicio y el final de viaje, desagregada en hora, minutos y segundos, además de la fecha. Construyamos una variable de tipo fecha-tiempo, en la que se registre el inicio y el final de cada viaje. Para ello usaremos la función **make_datetime()**.

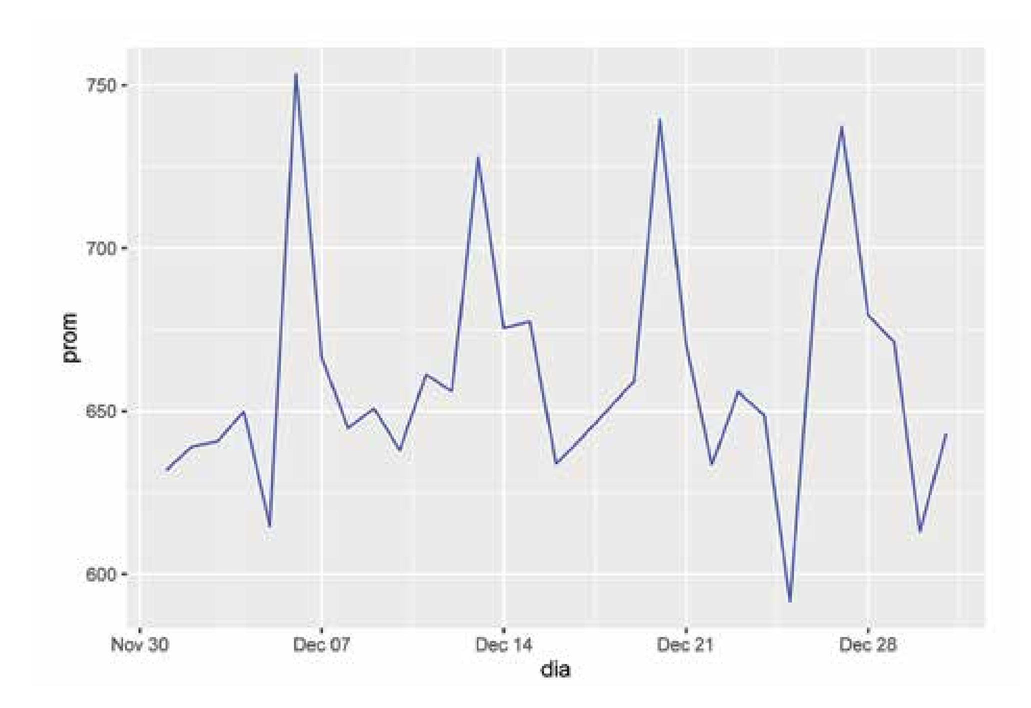
Después calculemos la duración de cada uno de los viajes. Para ello solo necesitamos hacer la resta entre el inicio y el final.

```
mibici2<-mibici %>%
     mutate(inicio=make_datetime(year, month, day, hour_inicio, minute_ini-
cio, se          cond_inicio))%>%
     mutate(final=make_datetime(year, month, day, hour_final, minute_final,
second_f         inal))%>%
     mutate(dura=final-inicio)
mibici2[1:5,16:18]
```

```
## # A tibble: 5 x 3
##    inicio                final                 dura
##    <dttm>                <dttm>                <drtn>
## 1  2020-12-22 18:09:53   2020-12-22 18:23:13   800 secs
## 2  2020-12-23 12:49:01   2020-12-23 13:03:34   873 secs
## 3  2020-12-15 18:29:50   2020-12-15 18:39:19   569 secs
## 4  2020-12-06 11:54:35   2020-12-06 12:13:44   1149 secs
## 5  2020-12-09 06:35:46   2020-12-09 07:03:21   1655 secs
```

Observamos que la duración de cada viaje se encuentra segundos. Con esta información podríamos calcular el promedio de la duración por día y graficarla. Para hacemos uso de las herramientas que hemos aprendido en capítulos anteriores sobre el manejo de datos y ggplot.

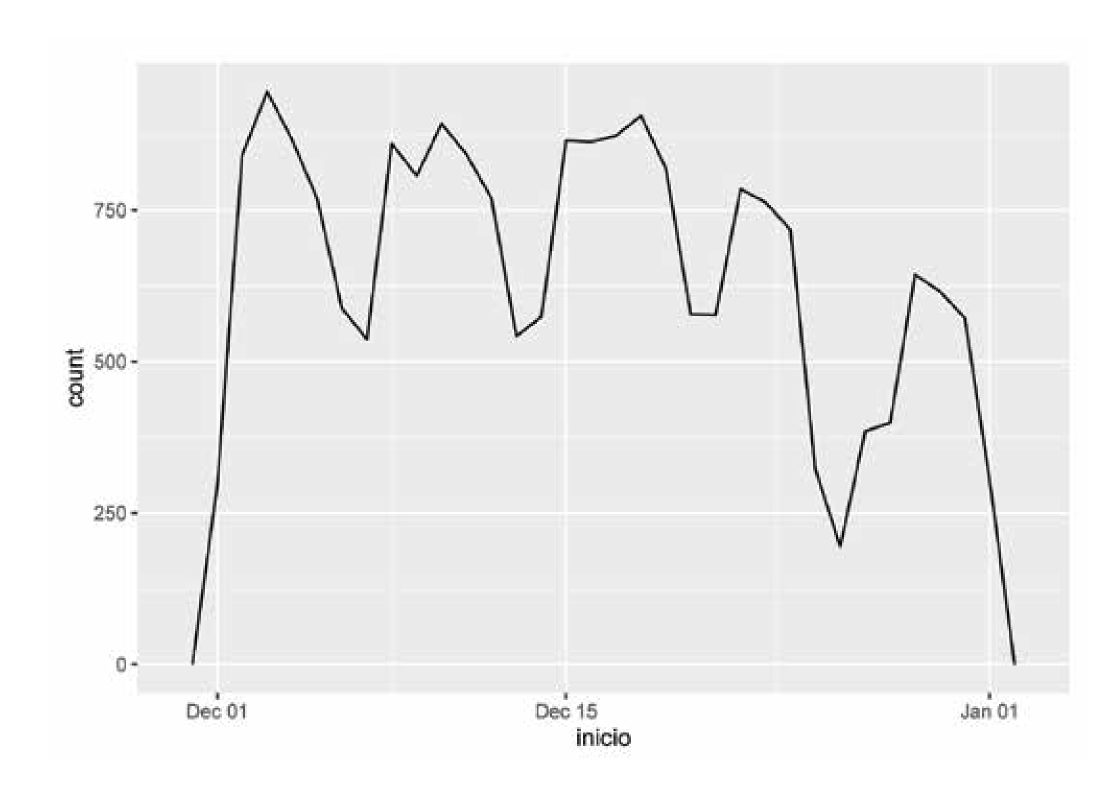
```
promedio<-summarise(group_by(mibici2,dia=make_datetime(year, month, day)),
prom=mean(dura), .groups = 'dropggplot(promedio, aes(dia, prom))+
     geom_line(col="blue")
```

Observa que hemos efectuado la agrupación por día, por ello la instrucción contempla **dia=make_datetime(year, month, day)**.

Podemos también visualizar el número de viajes por día. Para ello hacemos;

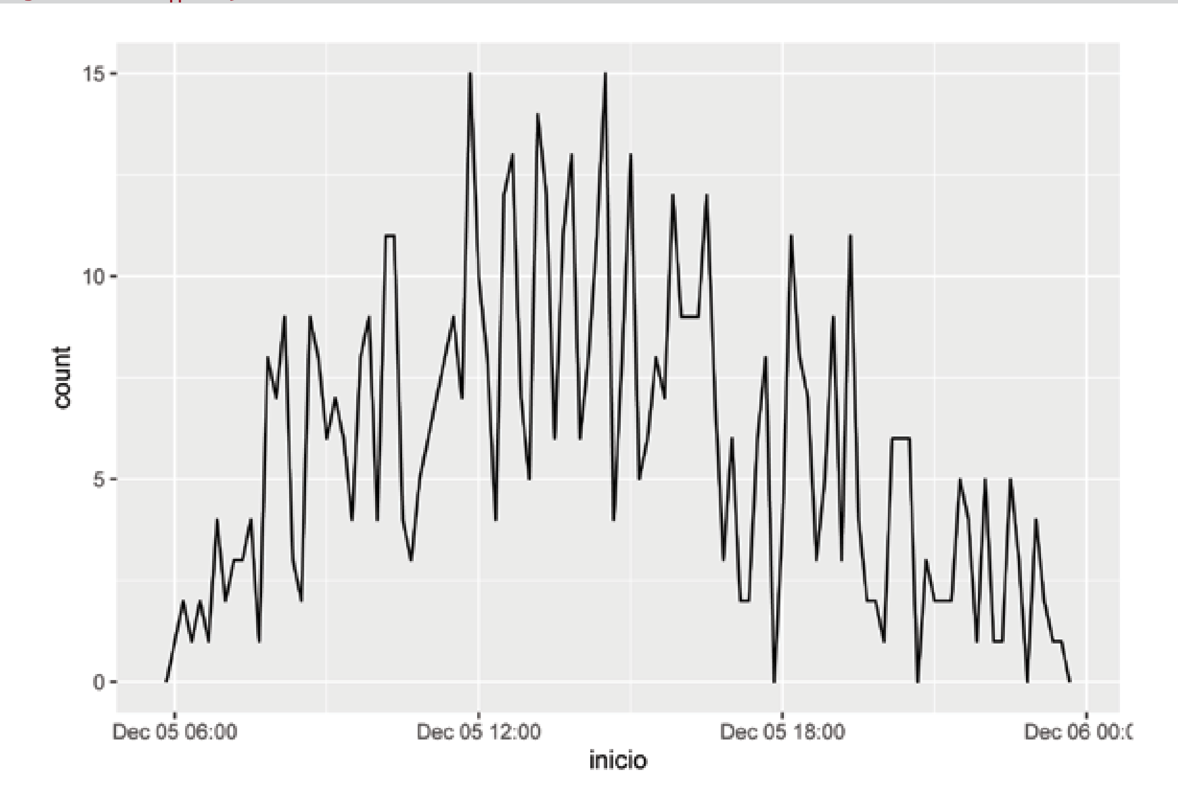
```
mibici2 %>%
     ggplot(aes(inicio))+
     geom_freqpoly(binwidth=86400)
```



Hemos indicado para el tamaño con 86400, el equivalente a un día en segundos.

Podemos graficar también la información de los viajes solo durante un día. Filtremos solo la información del día 5 y grafiquemos el comportamiento de los viajes cada 10 minutos (600 segundos).

```
mibici2 %>%
     filter(day==5) %>%
     ggplot(aes(inicio)) +
     geom_freqpoly(binwidth = 600)
```



3.3 Otras formas

Es posible hacer las siguientes conversiones

- De fecha a fecha-tiempo con **as_datatime()**
- De fecha-tiempo a fecha con **as_date()**

Por ejemplo; veamos el efecto de **as_datatime()** sobre la fecha del día de hoy.

```
today()
## [1] "2021-01-09"
```

```
class(today())
## [1] "Date"
```



```
as_datetime(today())
## [1] "2021-01-09 UTC"
```

```
class(as_datetime(today()))
## [1] "POSIXct" "POSIXt"
```

Observa que cambia la representación y agrega una unidad de tiempo UTC (Tiempo Coordinado Universal) En el caso de la función **as_date()** tenemos

```
now()
## [1] "2021-01-09 14:39:00 CST"
```

```
class(now())
## [1] "POSIXct" "POSIXt"
```

```
as_date(now())
## [1] "2021-01-09"
```

```
class(as_date(now()))
## [1] "Date"
```

Esto puede ser útil en caso de que tengamos una variable dentro de una base de datos que contenga horas y fechas y solo nos interesen las fechas, tal como el caso de la base de datos de *mibici*

4 Componentes de fechas y tiempos

Consideremos el primer viaje de la base de datos *mibici2*. Asignemos la fecha y la hora de inicio a una variable que llamaremos *fecha_prueba*

```
fecha_prueba<-mibici2$inicio[1]
fecha_prueba
## [1] "2020-12-22 18:09:53 UTC"
```

A través de diversas funciones podemos obtener los componentes de esta fecha.

```
year(fecha_prueba) #Año
## [1] 2020
month(fecha_prueba) #Mes
## [1] 12

month(fecha_prueba, label = TRUE, abbr = FALSE)
## [1] December
## 12 Levels: January < February < March < April < May < June < ... < December

mday(fecha_prueba) #Día
## [1] 22
yday(fecha_prueba) #Día del año
## [1] 357

wday(fecha_prueba) #Día de la semana
## [1] 3

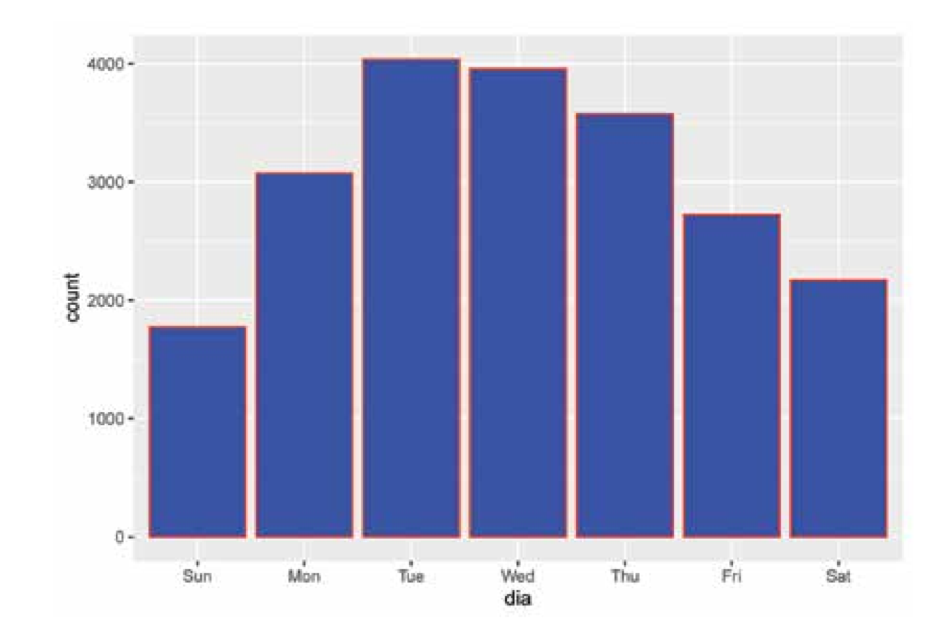
wday(fecha_prueba, label = TRUE, abbr = FALSE)
## [1] Tuesday
## 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday

wday(fecha_prueba, label = TRUE, abbr = TRUE)
## [1] Tue
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Mediante las opciones **label = TRUE, abbr = FALSE** podemos incluir la solicitud de que se regrese el nombre del mes o día, así como la abreviación de este. Estas opciones funcionan para **month()** y **wday()**.

Con estos elementos podemos analizar que día de la semana se reportaron mayor número de viajes.

```
mibici2 %>%
     mutate(dia=wday(inicio, label = TRUE)) %>%
     ggplot(aes(x=dia))+
     geom_bar(fill="blue", col="red")
```

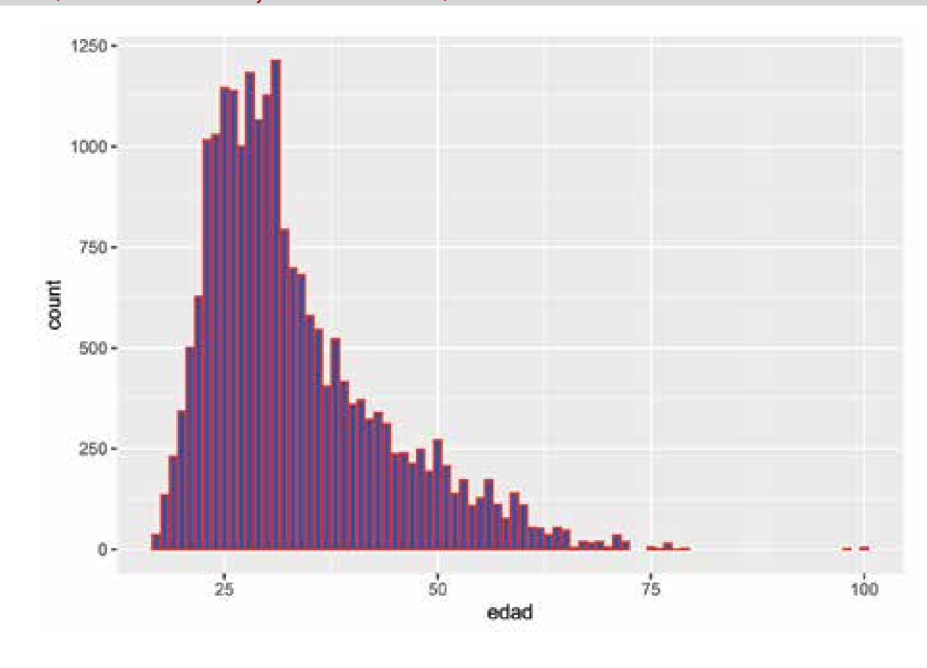



La indicación **dia=wday(inicio, label = TRUE)** genera una variable que incluye alguno de los días de la semana en relación a la fecha en la que se inició el viaje.

```
wday(fecha_prueba, label = TRUE)
## [1] Tue
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Dentro de la tabla de datos de mibici se incluye el año de nacimiento del usuario del servicio. Podemos calcular la edad del usuario y calcular el número de viajes que realizan los usuarios de esas edades.

```
mibici2 %>%
     mutate(edad=2020-nacimiento) %>%
     ggplot(aes(x=edad))+
     geom_bar(fill="blue", col="red")
```



4.1 Redondear fechas-tiempo

Una mas de las funcionalidades que ofrece **lubridate()** es la posibilidad de redondear fechas. Por ejemplo, supongamos que tenemos una fecha con un tiempo determinado (que incluye un año, un mes, un día, un minuto y un segundo específico) y que es de nuestro interés efectuar el redondeo a la unidad de tiempo mas cercana. Para ello disponemos de tres funciones;

- floor_date(): Redondea hacia la semana, el mes, el año, etc. anterior
- round_date(): Redondea hacia la semana, el mes, el año, etc. mas próximo
- ceiling_date(): Redondea hacia la semana, el mes, el año, etc. siguiente

```
fecha_prueba
## [1] "2020-12-22 18:09:53 UTC"
```

```
floor_date(fecha_prueba, "week")
## [1] "2020-12-20 UTC"
ceiling_date(fecha_prueba, "week")
## [1] "2020-12-27 UTC"
```

Observa que en el ejemplo la fecha de prueba que habíamos establecido anteriormente era 22 de diciembre. Al hacer el redondeo hacia la semana **(week)** con **floor_date()**, nos resulta 20 de diciembre correspondiente a la semana anterior, mientras que si lo hacemos con **ceiling_date()** lo hace hacia la semana siguiente. Consulta un calendario para que compruebes este hecho.

Podemos cambiar la especificación sobre la temporalidad sobre la que deseamos hacer el redondeo:

```
floor_date(fecha_prueba, "day")
## [1] "2020-12-22 UTC"
```

```
ceiling_date(fecha_prueba, "day")
## [1] "2020-12-23 UTC"
```

```
floor_date(fecha_prueba, "month")
## [1] "2020-12-01 UTC"
```

```
ceiling_date(fecha_prueba, "month")
## [1] "2021-01-01 UTC"
```



```
floor_date(fecha_prueba, "hour")
## [1] "2020-12-22 18:00:00 UTC"
```

```
ceiling_date(fecha_prueba, "hour")
## [1] "2020-12-22 19:00:00 UTC"
```

```
floor_date(fecha_prueba, "minute")
## [1] "2020-12-22 18:09:00 UTC"
```

```
ceiling_date(fecha_prueba, "minute")
## [1] "2020-12-22 18:10:00 UTC"
```

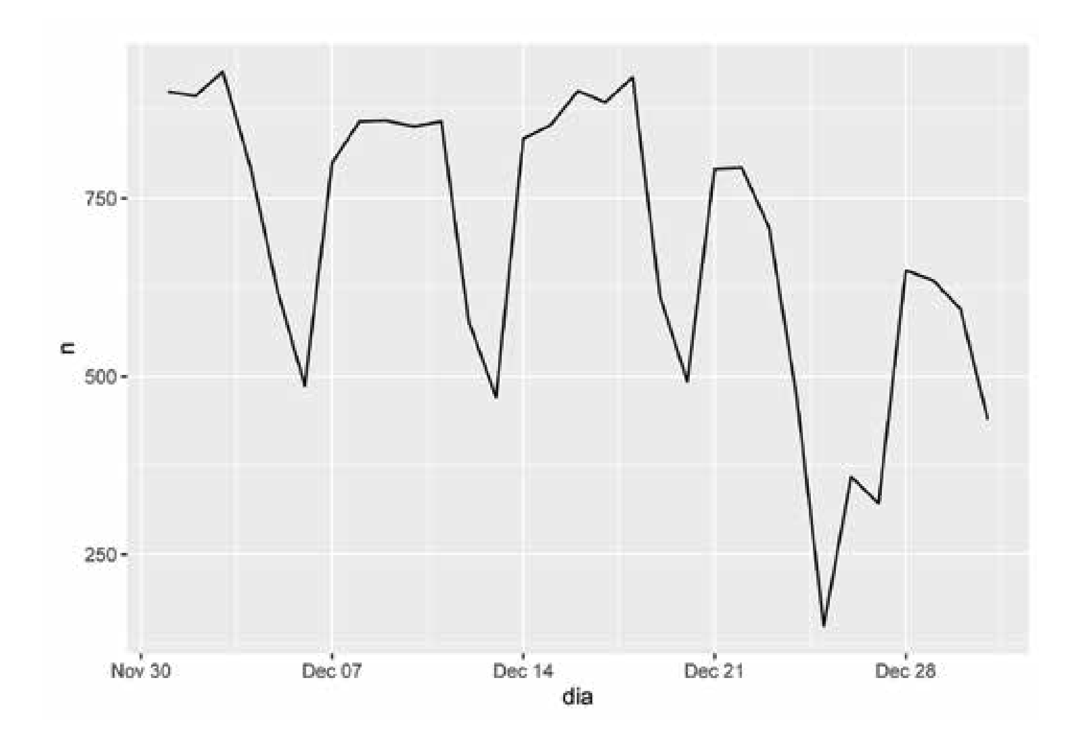
También podemos usar la función **round_date()** que efectúa el redondeo hacia el momento mas cercano en relación a la unidad de tiempo.

```
round_date(fecha_prueba, unit = "week")
## [1] "2020-12-20 UTC"
```

```
round_date(fecha_prueba, unit = "minute")
## [1] "2020-12-22 18:10:00 UTC"
```

Haciendo uso de estos elementos, podemos contabilizar los viajes, considerando el día en que se iniciaron.

```
mibici2 %>%
     count(dia=floor_date(inicio, "day")) %>%
     ggplot(aes(dia,n))+
     geom_line()
```



En la instrucción anterior hemos indicado que se cree una variable de día y se contabilicen los viajes efectuados cada día. Esto gracias a la función **count()**.

4.2 Actualizar fecha-tiempo

Si deseamos cambiar alguno de los componentes de una fecha, podemos usar la función **update()**

```
fecha_prueba
```

```
## [1] "2020-12-22 18:09:53 UTC"
```

```
#Cambiar el mes
update(fecha_prueba, month=11)
```

```
## [1] "2020-11-22 18:09:53 UTC"
```

```
#Cambiar el día
update(fecha_prueba, mday=20)
```

```
## [1] "2020-12-20 18:09:53 UTC"
```

```
#Cambiar la hora
update(fecha_prueba, hour=15)
```

```
## [1] "2020-12-22 15:09:53 UTC"
```


5 Lapsos de Tiempo

Como te habrás dado cuenta fue posible calcular la duración de cada viaje, haciendo la resta entre la hora de inicio y la hora de termino del viaje. En **R** es posible calcular lapsos de tiempo como; duración, periodos o intervalos. Esto representa la aritmética de fechas y tiempos.

Ya hemos efectuado operaciones sobre la duración y vimos que el resultado se expresó en segundos. Tomemos la información de inicio y fin del primer viaje de la tabla o base de datos de mibici.

```
inicio<-mibici2$inicio[1]
inicio
```

```
## [1] "2020-12-22 18:09:53 UTC"
```

```
fin<-mibici2$final[1]
fin
```

```
## [1] "2020-12-22 18:23:13 UTC"
```

Sabemos que la duración es

```
viaje<-fin-inicio
as.duration(viaje)
```

```
## [1] "800s (~13.33 minutes)"
```

La función as.duration nos regresa la duración de este primer lapso. El resultado se expresa en segundos, pero incluye una conversión a minutos.

Podemos calcular la duración en segundos de lapsos de tiempo;

```
dminutes(c(13,15))
```

```
## [1] "780s (~13 minutes)" "900s (~15 minutes)"
```

```
dhours(2.5)
```

```
## [1] "9000s (~2.5 hours)"
```

```
ddays(1:3)
```

```
## [1] "86400s (~1 days)" "172800s (~2 days)" "259200s (~3 days)"
```

```
dweeks(2)
```

```
## [1] "1209600s (~2 weeks)"
```

```
dyears(.5)
```

```
## [1] "15778800s (~26.09 weeks)"
```

En el primer caso con dminutes hemos pedido conocer cuantos segundos hay en trece minutos y luego en quince. En el segundo caso con dhours, los segundos de 2.5 horas. El tercer caso mediante **ddays** solicitamos los segundos de 1,2 y 3 días.

Cada una de estas funciones regresa la duración en segundos, de valores en minutos, horas, días, semanas o años.

Con estas funciones podemos efectuar operaciones sobre el tiempo. Por ejemplo, la siguiente indicación calcula los segundos en cinco minutos y los multiplica por 3.

```
3*dminutes(5)
```

```
## [1] "900s (~15 minutes)"
```

Mediante la operación de estas funciones, podemos construir variables que representen fechas ;

```
today() #hoy
```

```
## [1] "2021-01-09"
```

```
today()+ddays(1) #mañana
```

```
## [1] "2021-01-10"
```

```
today()+dyears(1) #el siguiente año
```

```
## [1] "2022-01-09 06:00:00 UTC"
```

La duración con estas funciones contabiliza lapsos con un número determinado de segundos.

Es posible cambiar los lapsos de tiempo que se contabilizan en segundos, usando periodos de tiempo. Para ello usamos las funciones;

```
minutes(c(13,15))
```

```
## [1] "13M 0S" "15M 0S"
```

```
hours(3)
```

```
## [1] "3H 0M 0S"
```



```
days(1:3)
```

```
## [1] "1d 0H 0M 0S" "2d 0H 0M 0S" "3d 0H 0M 0S"
```

```
weeks(2)
```

```
## [1] "14d 0H 0M 0S"
```

```
years(5)
```

```
## [1] "5y 0m 0d 0H 0M 0S"
```

Observa qué en este caso, los resultados muestran periodos de tiempo, en lugar de lapsos (duración) en segundos.

Mediante el uso de la función **seconds_to_period** es posible transforma un valor númerico en segundo, en un periodo de tiempo. Por ejemplo 13289 segundos equivalen a 3 horas, 41 minutos y 29 segundos.

```
seconds_to_period(13289)
```

```
## [1] "3H 41M 29S"
```

El uso de esta función proporciona un estimación exacta, ya que al expresar los resultados en términos de días, horas, minutos y segundos, no involucra el uso de años o meses. Podemos tambien hacer uso de la función contraria, la cual regresará el número de segundos en un periodo.

```
periodo<-seconds_to_period(13289)
periodo
```

```
## [1] "3H 41M 29S"
```

```
period_to_seconds(periodo)
```

```
## [1] 13289
```

Con estas funciones también es posible efectuar operaciones que generen nuevos periodos. Por ejemplo;

```
days(7)+months(3)+seconds(10)
```

```
## [1] "3m 7d 0H 0M 10S"
```

```
2*days(7)+months(3)+seconds(10)
```

```
## [1] "3m 14d 0H 0M 10S"
```

Con las funciones que hemos revisado hasta el momento, esperaríamos que al efectuar la división **years(1) / days(1)** obtuviéramos un 365. Sin embargo, el efectuar la siguiente operación obtenemos;

```
years(1) / days(1)
```

```
## [1] 365.25
```

Resultando no precisamente en 365, pues se trata de una aproximación basada en la contabilización con segundos. Si deseemos una mejor aproximación tenemos que hacer un intervalo de un año, para construirlo debemos apoyarnos de %--%

```
sig_a<-today()+years(1)
today() %--% sig_a #Este es el intervalo exacto de un año
```

```
## [1] 2021-01-09 UTC--2022-01-09 UTC
```

```
(today() %--% sig_a) /days(1)
```

```
## [1] 365
```

```
now() %--% (now()+hours(1)) #Este in intervalo exacto de una hora
```

```
## [1] 2021-01-09 14:39:04 CST--2021-01-09 15:39:04 CST
```

```
now() %--% (now()+seconds(1))#Este in intervalo exacto de un segundo
```

```
## [1] 2021-01-09 14:39:04 CST--2021-01-09 14:39:05 CST
```

6 Zonas horarias

Como sabes existen diferentes zonas horarias, lo cual puede convertirse en un problema en el momento en que trabajamos con fechas y tiempo. De manera automática cuando descargas **R** por primera vez en tu computadora, se configura una zona horaria. Para determinar cual es la zona horaria establecida usamos;

```
Sys.timezone()
```

```
## [1] "America/Mexico_City"
```

Si por alguna razón no existe una zona horaria configurada, esta instrucción regresará *NA*.

Podemos configurar *R* en determina zona horaria haciendo:

```
Sys.setenv(tz="Europe/Madrid")
now()
```

```
## [1] "2021-01-09 14:39:04 CST"
```

Esto fijará la zona horaria de Europa.

Regresando a la zona horaria de México;

```
Sys.setenv(tz="America/Mexico_City")
now()
```

```
## [1] "2021-01-09 14:39:04 CST"
```

Para consultar las zonas horarias disponibles, podemos usar la función **OlsonNames()**

```
head(OlsonNames())
```

```
## [1] "Africa/Abidjan" "Africa/Accra" "Africa/Addis_Ababa"
## [4] "Africa/Algiers" "Africa/Asmara" "Africa/Asmera"
```

La función **head()**muestra los primeros elementos.

En caso de que ya tengamos una variable que contenga una fecha en un determinado formato de zona horaria, podemos cambiarla;

```
fecha_prueba
```

```
## [1] "2020-12-22 18:09:53 UTC"
```

```
fecha_prueba2<-with_tz(fecha_prueba, tz="Europe/Madrid")
fecha_prueba2
```

```
## [1] "2020-12-22 19:09:53 CET"
```

Esto simplemente es para representar el **mismo instante** en diferentes horarios. Esto lo sabemos pues si hacemos la resta entre las dos fechas tenemos una diferencia de cero segundos;

```
fecha_prueba-fecha_prueba2
```

```
## Time difference of 0 secs
```

Otro ejemplo;

```
instante<-ymd_hms("2015-06-01 12:00:00", tz = "Europe/Madrid") #Un instante
en la zona de America/New_York
instante2<-with_tz(instante, tz="America/Mexico_City") #El mismo instante en
la zona "America/Mexico_City"
instante2
```

```
## [1] "2015-06-01 05:00:00 CDT"
```

```
instante-instante2
```

```
## Time difference of 0 secs
```

En caso de que un instante (fecha o tiempo) se encuentre mal identificado en su zona horaria, debemos forzar el cambio de zona horaria. Para ello usamos **force_tz()**

```
instante3<-force_tz(instante, tz="America/Mexico_City")
instante3
```

```
## [1] "2015-06-01 12:00:00 CDT"
```

```
instante2- instante3
```

```
## Time difference of -7 hours
```

Forzando al **instante** a que se encuentre en la zona distinta a la que era inicialmente, veamos que ahora si hay diferencia en el tiempo.

Es de nuestro interes conservar unicamente el valor númerico contendio en el resultado anterior, podemos hacer;

```
as.numeric(instante2- instante3)
```

```
## [1] -7
```

7 Actividades

1. Usa la función adecuada para convertir las siguientes cadenas de texto en fechas;

- "March 8, 2021"
- "2021-June-07"
- "06-April-2019"

- c("August 19 (2015)", "July 1 (2015)")
- "12/30/20" 30 de diciembre 2020

2. Usa la tabla de datos de mibici y construye una gráfica donde se muestre el número de viajes por día, usando únicamente los viajes que duraron más de 20 minutos.

3. Escribe una función que dada una fecha de nacimiento exacta (con horas, minutos y segundos), pueda calcular la edad y expresarla como un periodo en términos de días, horas y segundos.

4. Determina que día de la semana será el 05 de junio de 2021.

5. Construye un periodo que tenga 5 años, 3 meses, 2 días, 15 horas, 10 minutos y 5 segundos. Luego averigua la duración del periodo en segundos.

Capítulo 10

Factores

Capítulo 10 | Factores

1 Introducción

Un factor en R se refiere a una variable categórica, la cual tiene un número finito y conocido de valores posibles. Los factores son útiles para mostrar variables de texto en un orden personalizado, diferente al orden alfabético y también se utilizan para cambiar el valor de esas variables.

2 Forcats

Para trabajar factores es necesaria la librería forcats. Ya que no forma parte del conjunto tidyverse, hay que instalarla y cargarla explícitamente. Además, recordemos iniciar nuestro script de la forma usual, cargando el directorio donde se encuentre la base de datos que trabajaremos en este capítulo.

```
setwd("~/Dropbox/Curso de R/Cap10_Factores")
options(scipen=999)
library (tidyverse)
library(readr)
library(forcats)
```

En este capítulo utilizaremos una muestra de la edición de 2018/2019 de las encuestas de opinión pública en las Américas (LAPOP) en México, que realiza la Universidad Vanderbilt. Estos datos se encuentran en el excel en el archivo de nombre lapop_mexico.csv, una vez cargada la base de datos, cambiemos los nombres de las siguientes variables para facilitar el manejo de los datos de este capítulo, ya que algunos nombres que han sido asignados no se relacionan con el contenido:


```
lapop <- read_csv("lapop_mexico.csv")
colnames(lapop)
```

```
## [1]  "idnum"        "uniq_id"        "cluster"        "upm"        "wt"
## [6]  "wave"         "pais"       "nationality"  "estratopri"     "prov"
## [11] "municipio"    "estratosec"     "tamano"         "ur"         "fecha"
## [16] "q1"           "q2"             "a4"             "vb1"        "inf1"
## [21] "vb2"          "vb3n"           "vb10"           "vb11"       "pol1"
## [26] "vb20"         "q10new"
```

```
names(lapop)[9:10] <- c("region", "entidad")
names(lapop)[16:18] <- c("sexo", "edad", "problema_mas_grave")
names(lapop)[22] <- "voto"
names(lapop)[25:27] <- c("confianza_politica", "elecciones", "ingreso")
```

Observa que para cambiar el nombre de la columna hemos indicado el número de columna seguido del nombre que deseamos asignar en el caso en que se trate de columnas sucesivas hemos incluido los dos puntos. Puede volver a ejecutar **colnames(lapop)** para verificar que los nombres han cambiado.

Cuando importamos una base de datos, **R** no siempre es capaz de identificar cuando una variable es solo texto o cuando es un factor. Veamos un resumen de la base **lapop** para el ver tipo de variables.

```
summary(lapop)
```

```
##     idnum            uniq_id            cluster            upm
##     Min.   : 1.0     Length:1580        Min.   :  11.0     Min.   :  1.00
##     1st Qu.: 403.8   Class :character   1st Qu.: 332.0     1st Qu.: 32.00
##     Median : 811.5   Mode  :character   Median : 652.0     Median : 65.00
##     Mean   : 810.9                      Mean   : 656.6     Mean   : 65.11
##     3rd Qu.:1212.2                      3rd Qu.: 981.2     3rd Qu.: 98.00
##     Max.   :1624.0                      Max.   :1302.0     Max.   :130.00
##           wt           wave         pais               nationality
##     Min.   :1    Min.   :2018     Length:1580        Length:1580
##     1st Qu.:1    1st Qu.:2018     Class :character   Class :character
##     Median :1    Median :2018     Mode  :character   Mode  :character
##     Mean   :1    Mean   :2018
##     3rd Qu.:1    3rd Qu.:2018
```

```
##    Max. :1    Max. :2018
##    region          entidad          municipio        estratosec
##    Length:1580     Length:1580      Length:1580      Length:1580
##    Class :character Class :character Class :character Class :character
##    Mode :character  Mode :character  Mode :character  Mode :character
##
##
##    tamano          ur               fecha                sexo
##    Length:1580     Length:1580      Length:1580          Length:1580
##    Class :character Class :character Class :character     Class :character
##    Mode :character  Mode :character  Mode :character      Mode :character
##
##
##
##    edad            problema_mas_grave    vb1             inf1
##    Min. :18.00     Length:1580           Length:1580     Length:1580
##    1st Qu.:28.00   Class :character      Class :character Class :character
##    Median :40.00   Mode :character       Mode :character  Mode :character
##    Mean :42.09
##    3rd Qu.:54.25
##    Max. :88.00
##    vb2             voto             vb10             vb11
##    Length:1580     Length:1580      Length:1580      Length:1580
##    Class :character Class :character Class :character Class :character
##    Mode :character  Mode :character  Mode :character  Mode :character
##
##
##    confianza_politica    elecciones            ingreso
##    Length:1580           Length:1580           Length:1580
##    Class :character      Class :character      Class :character
##    Mode :character       Mode :character       Mode :character
##
##
##
```


La mayoría de las variables son reconocidas como texto. Sin embargo necesitamos que sean factores (categóricas). Por esta razón debemos transformarlas, ya que el conjunto posible de respuestas es conocido.

Para ello utilizamos la función `as.factor()`.

```
lapop <- lapop %>%
   mutate(region=as.factor(region), sexo=as.factor(sexo),
      entidad=as.factor(entidad), problema_mas_grave=as.factor(problema_mas_
grave),
      voto=as.factor(voto),confianza_politica=as.factor(confianza_politica),
      elecciones=as.factor(elecciones),ingreso=as.factor(ingreso))
```

Con esto las variables indicadas ahora son del tipo factor. Te invito a que vuelvas a efectuar el **summary(lapop)** y observes las diferentes antes y después de convertir determinadas variables de la base en factor.

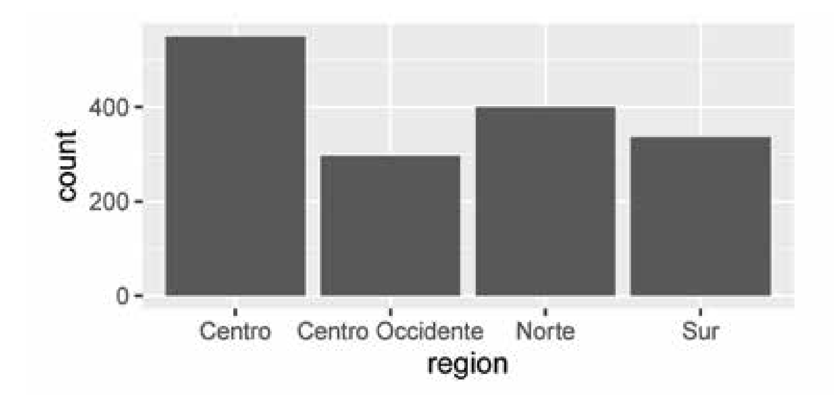
Cuando trabajamos con este tipo de respuestas, si existen muchas opciones para una variable, podemos obtener un conteo rápido, tal como lo hemos hecho con anterioridad:

```
unique(lapop$region)
## [1]      Centro     Centro Occidente Norte      Sur
## Levels: Centro     Centro Occidente Norte Sur
```

```
lapop %>%
     count(region)
## # A tibble: 4 x 2
##    region           n
##    <fct>            <int>
## 1  Centro           549
## 2  Centro Occidente 296
## 3  Norte            399
## 4  Sur              336
```

También podemos verlo a través de una gráfica de barras:

```
ggplot(lapop, aes(region))+
     geom_bar()
```

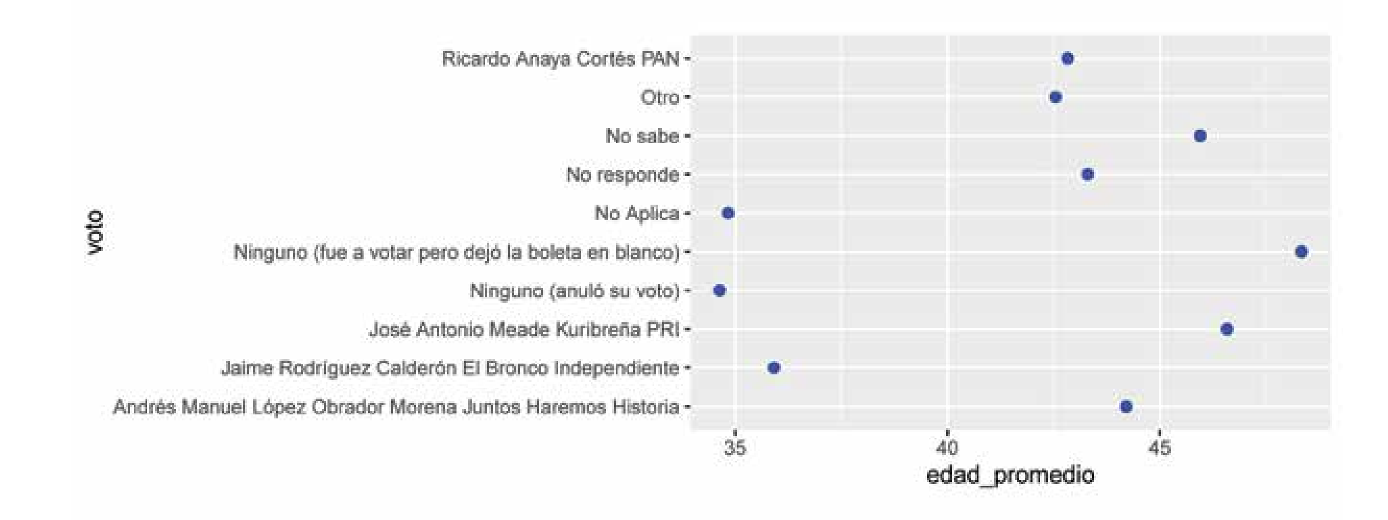


Cuando trabajemos con factores nos enfrentaremos a dos problemas comunes. El primero es la necesidad de cambiar el orden de las respuestas y el según es cambiar el contenido de las respuestas o el valor de las respuestas por uno más conveniente.

3 Modificar el orden de los factores

La variable voto reporta el voto de los ciudadanos en las elecciones presidenciales pasadas, e imaginemos que nos interesa analizar la edad de quienes votaron por los diferentes candidatos:

```
voto <- lapop %>%
     group_by(voto) %>%
     summarize(edad_promedio = mean(edad, na.rm = TRUE), n = n(),
.groups='drop')
     ggplot(voto, aes(edad_promedio, voto)) +
     geom_point(size=2, col="blue")
```

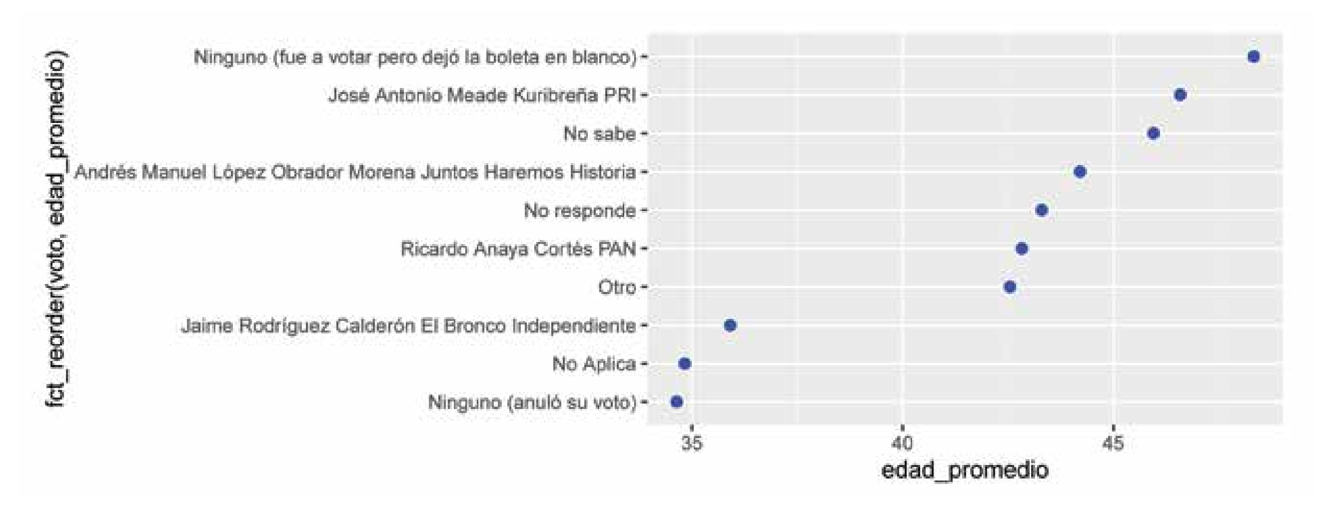


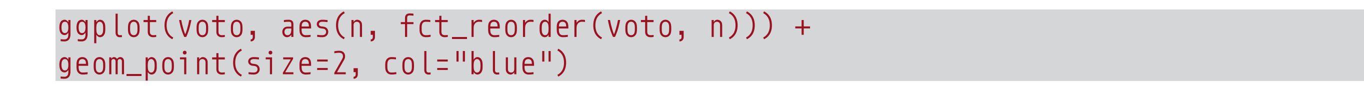
La gráfica nos muestra el promedio de edad según el voto realizado, sin embargo, podemos mejorar la visualización si reordenamos los factores usando la instrucción *fct_reorder(),* esta instrucción utiliza tres argumentos:

- *f*, la variable factor que deseamos modificar
- *x*, una variable numérica que se usará para reordenar los factores
- *fun*, una opción que se usa cuando existen múltiples valores de x, utiliza por defecto la mediana

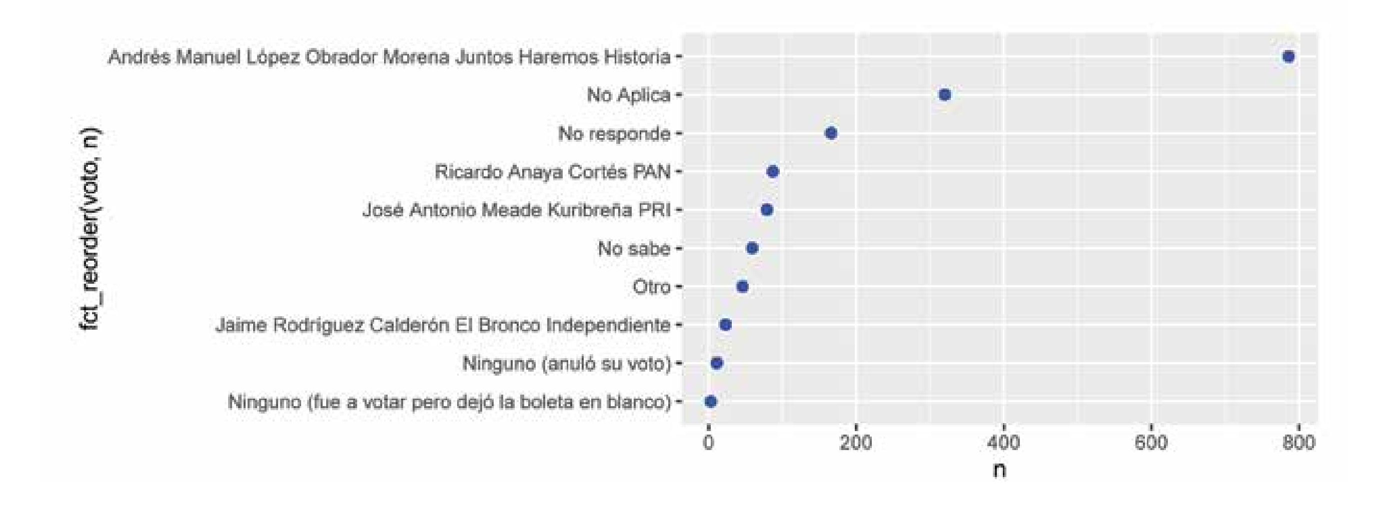
Por ejemplo, si reordenamos a los candidatos por la edad promedio de quienes reportaron sus votos, o si los reordenamos por el número de personas que reportaron votar por ellos, obtenemos que en ambas visualizaciones se vuelve más sencilla su interpretación.

```
ggplot(voto, aes(edad_promedio, fct_reorder(voto, edad_promedio))) +
geom_point(size=2, col="blue")
```





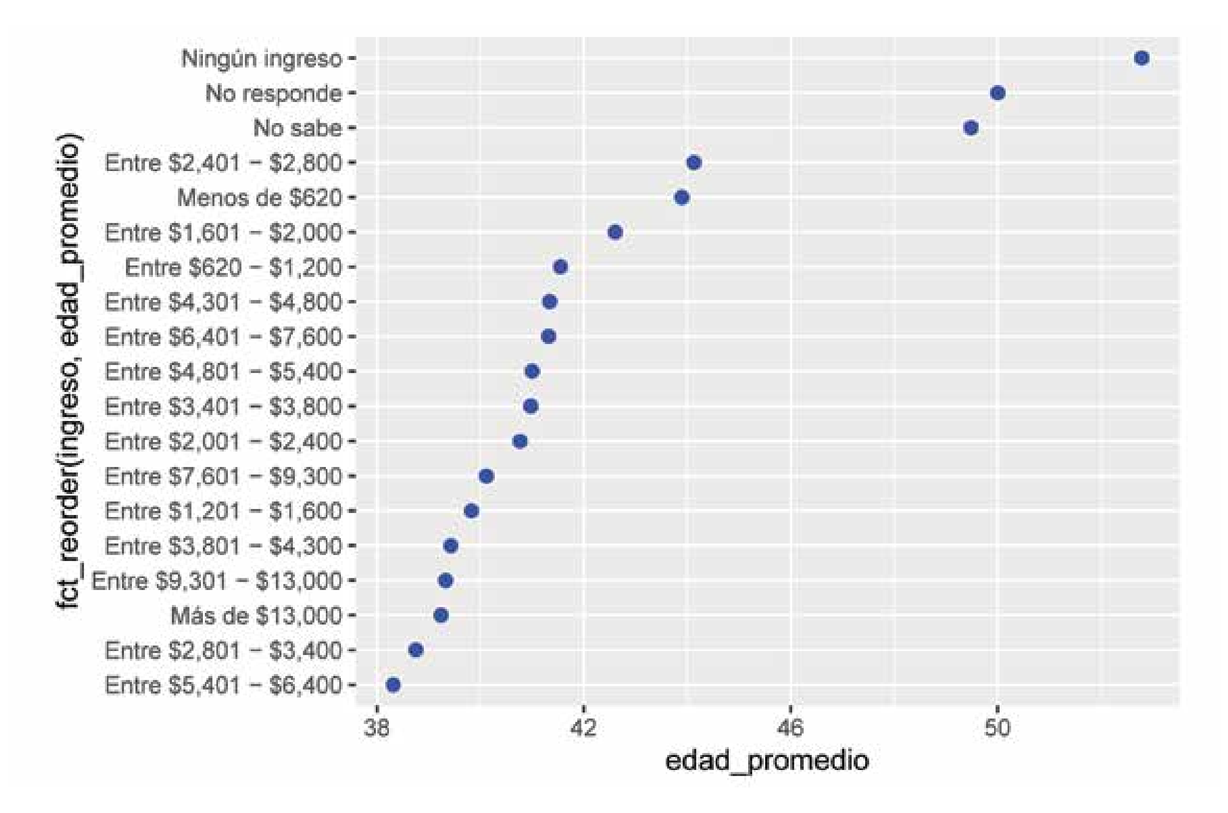
```
ggplot(voto, aes(n, fct_reorder(voto, n))) +
geom_point(size=2, col="blue")
```



Ahora analicemos la edad promedio según los niveles de ingreso, para ello obtenemos el promedio de la edad de quienes reportan tener determinado nivel de ingresos.

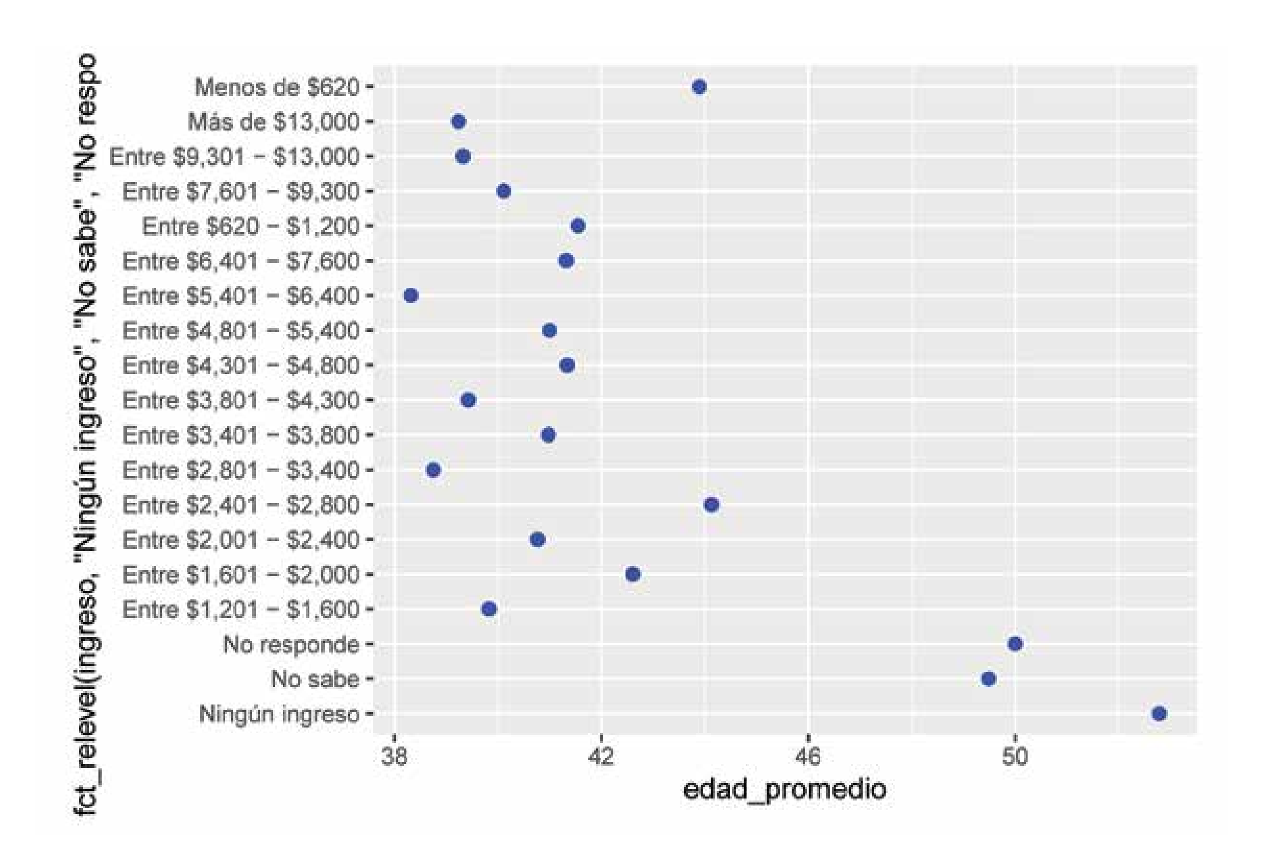
```
ingreso <- lapop %>%
     group_by(ingreso) %>%
     summarize(edad_promedio = mean(edad, na.rm = TRUE),n = n(),
.groups='drop' )

     ggplot(ingreso, aes(edad_promedio, fct_reorder(ingreso, edad_prome-
dio)))+                geom_point(size=2, col="blue")
```



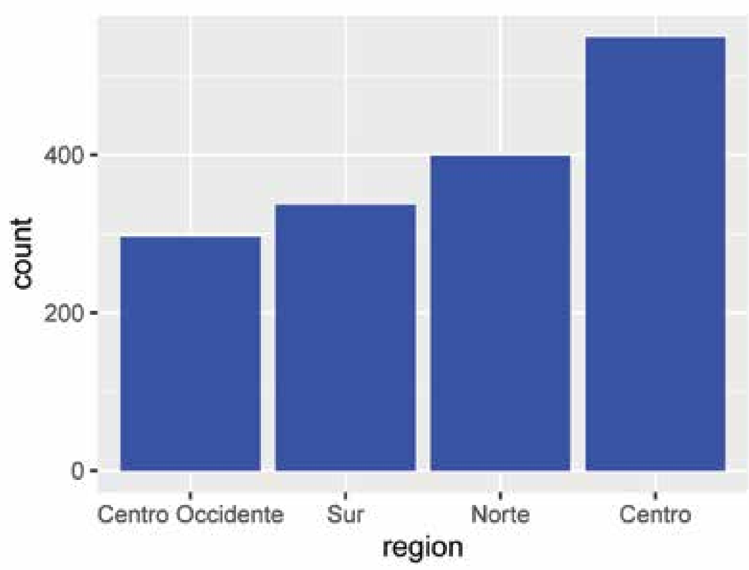
Sin embargo, en este caso es necesario aclarar que el ingreso ya tiene un orden predefinido de menor a mayor, por lo que reordenarlos por la edad promedio no necesariamente representa una mejora en la visualización. Una alternativa podría ser mover las opciones que representan un valor perdido, al final de las opciones, usando la siguiente instrucción:

```
ggplot(ingreso,
     aes(edad_promedio, fct_relevel(ingreso, "Ningún ingreso", "No sabe", "No
respon           de"))) +
geom_point(size=2, col="blue")
```

Otra forma de ordenar los factores es usando la opción *fct_infreq*, la cual reordena los factores por sus frecuencias de manera descendente, adicionalmente, podemos utilizar *fct_rev*, la cual nos permite invertir el orden y que sea de manera ascendente:

```
lapop %>%
mutate(region = region %>% fct_infreq() %>% fct_rev()) %>%
ggplot(aes(region)) +
geom_bar(fill="blue")
```



4 Modificar factores

El segundo problema, es cuando los factores tienen valores que resultan poco prácticos de analizar, por ejemplo, en las gráficas anteriores las posibilidades que tiene la variable voto, tienen nombres demasiado largos, lo que dificulta la visualización.

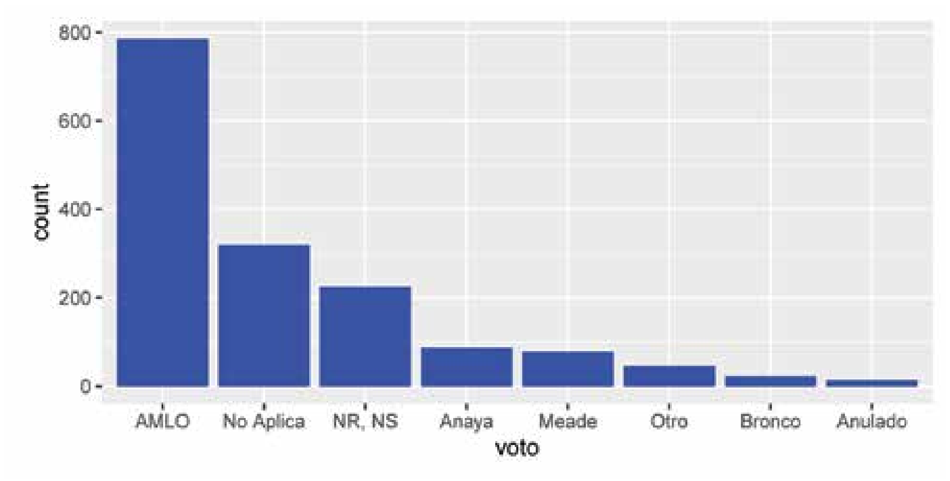
Para resolver este problema, necesitamos recodificar los valores de la variable voto, es decir, reemplazar las respuestas por otros nombres que sean más sencillos de analizar. Las visualizaciones útiles necesitan tener nombres cortos e informativos sobre las variables de interés.

Para recodificar una variable de tipo factor necesitamos utilizar la función *fct_recode()*, asignando la variable, seguido del nuevo valor que se asignará cuando se cumpla cierto otro valor:

```
lapop2 ← lapop %>%
mutate(voto = fct_recode(voto,
"AMLO" = "Andrés Manuel López Obrador Morena Juntos Haremos Historia",
"Bronco" = "Jaime Rodríguez Calderón El Bronco Independiente",
"Meade" = "José Antonio Meade Kuribreña PRI",
"Anaya" = "Ricardo Anaya Cortés PAN",
"NR, NS" = "No responde",
"NR, NS" = "No sabe",
"Anulado" = "Ninguno (anuló su voto)",
"Anulado" = "Ninguno (fue a votar pero dejó la boleta en blanco)"
))
```

Si graficamos de nuevo el voto de los participantes observamos que la visualización mejora considerablemente:

```
lapop2 %>%
mutate(voto = voto %>% fct_infreq()) %>%
ggplot(aes(voto)) +
geom_bar(fill="blue")
```



Observa que las opciones "No responde" y "No sabe" se agruparon en una misma categoría y las respuestas "Ninguno (anuló su voto)" y "Ninguno (fue a votar pero dejó la boleta en blanco)" también se agruparon en una misma respuesta. Cuando tenemos un número mayor de valores que se pueden resumir en grupos más pequeños puede resultar más útil utilizar la opción **fct_collapse()**, donde a cada nueva respuesta se le asignan resultados anteriores, veamos un ejemplo:

```
lapop ← lapop %>%
     mutate(voto = fct_recode(voto,
     "AMLO" = "Andrés Manuel López Obrador Morena Juntos Haremos Historia",
     "Bronco" = "Jaime Rodríguez Calderón El Bronco Independiente",
     "Meade" = "José Antonio Meade Kuribreña PRI",
     "Anaya" = "Ricardo Anaya Cortés PAN",)) %>%
     mutate(voto = fct_collapse(voto,
     "NR, NS" = c("No responde", "No sabe"),
     Anulado = c("Ninguno (anuló su voto)", "Ninguno (fue a votar pero dejó
la boleta         en blanco)")))
```

Otra forma de agrupar las respuestas es utilizando la opción *fct_lump()*, la cual es útil cuando tenemos opciones con pocas frecuencias y las agrupa para reducir el número de opciones y mejorar las visualizaciones, por ejemplo, en la encuesta se les pregunta a los ciudadanos sobre cuál considera que es el problema más grave en el país, si hacemos un conteo de las respuestas observamos que existen muchas opciones con muy pocas menciones:

```
unique(lapop$problema_mas_grave)
```

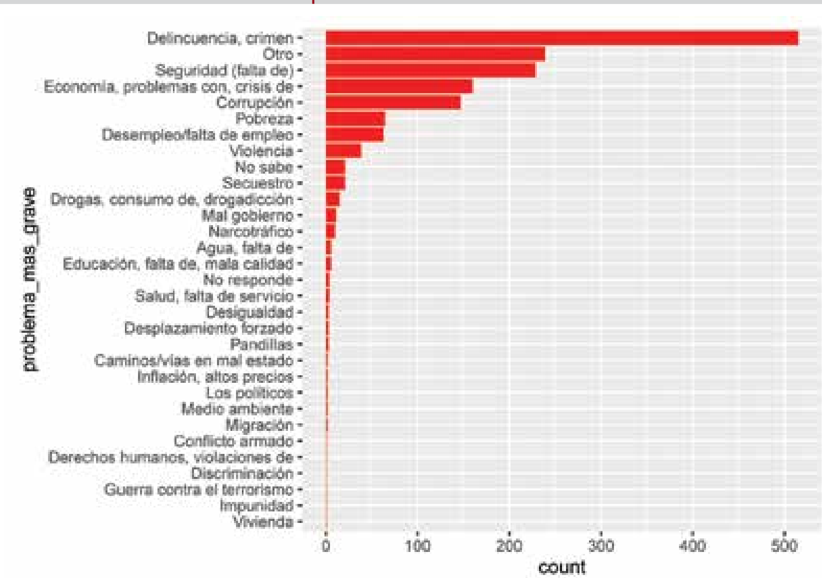
```
## [1] Otro                                  Violencia
## [3] Delincuencia, crimen                  Corrupción
## [5] Economía, problemas con, crisis de    Seguridad (falta de)
## [7] Pobreza                               Narcotráfico
## [9] Desempleo/falta de empleo             Secuestro
## [11] Inflación, altos precios             No sabe
## [13] Educación, falta de, mala calidad    Drogas, consumo de, drogadicción
## [15] Mal gobierno                         Desigualdad
## [17] Agua, falta de                       Conflicto armado
## [19] Salud, falta de servicio             Desplazamiento forzado
## [21] Migración                            Derechos humanos, violaciones de
## [23] Los políticos                        Pandillas
## [25] Caminos/vías en mal estado           No responde
## [27] Vivienda                             Guerra contra el terrorismo
```

```
## [29] Medio ambiente                       Impunidad
## [31] Discriminación
## 31 Levels: Agua, falta de Caminos/vías en mal estado ... Vivienda
```

```
lapop %>%
     count(problema_mas_grave)
```

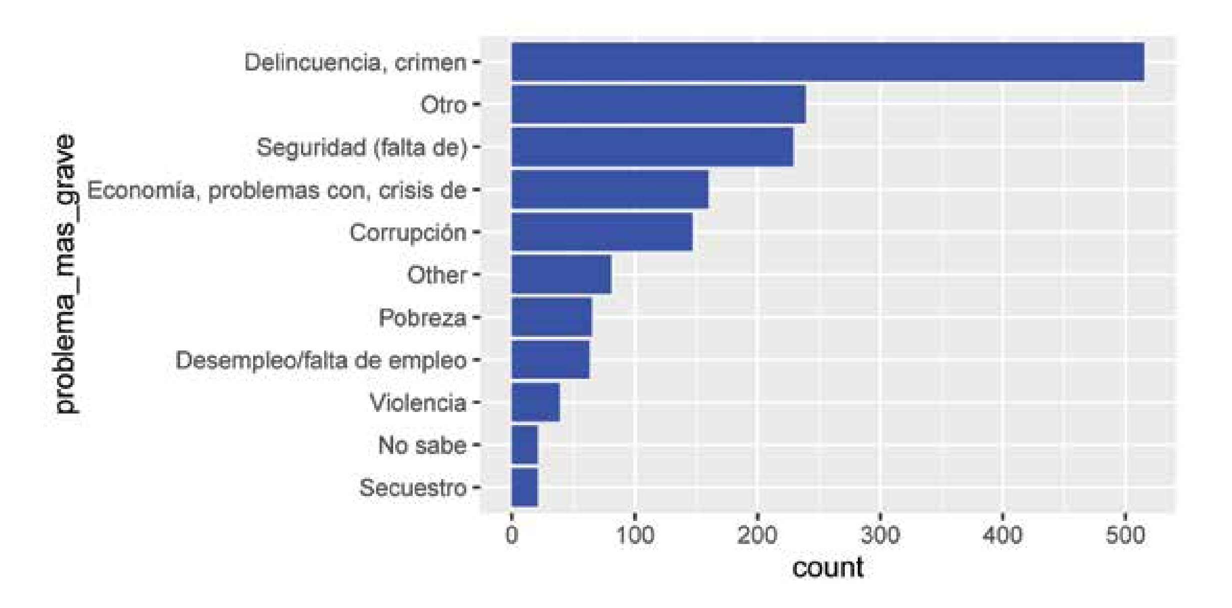
```
## # A tibble: 31 x 2
##    problema_mas_grave                 n
##    <fct>                              <int>
## 1  Agua, falta de                     6
## 2  Caminos/vías en mal estado         2
## 3  Conflicto armado                   1
## 4  Corrupción                         147
## 5  Delincuencia, crimen               515
## 6  Derechos humanos, violaciones de   1
## 7  Desempleo/falta de empleo          63
## 8  Desigualdad                        3
## 9  Desplazamiento forzado             3
## 10 Discriminación                     1
## # ... with 21 more rows
```

```
lapop %>%
mutate(problema_mas_grave = problema_mas_grave %>% fct_infreq() %>% fct_
rev()) %>%
ggplot(aes(problema_mas_grave)) +
geom_bar(fill="red") + coord_flip()
```



Ahora para reducir de manera automática las opciones con menor número de menciones utilizamos la *fct_lump(),* para utilizarla solo introducimos dentro del paréntesis la variable que deseamos agrupar, adicionalmente,utilizamos la opción *n=* para indicar el número de respuestas que deseamos conservar, así, si queremos solo las principales 10 respuestas, se conservaran las 10 respuestas con mayor número de frecuencias y el resto se agrupan en una nueva categoría, de manera predeterminada se agrupan con la etiqueta "Other".

```
lapop %>%
mutate(problema_mas_grave = fct_lump(problema_mas_grave, n=10) %>% fct_in-
freq()
%>% fct_rev()) %>%
ggplot(aes(problema_mas_grave)) +
geom_bar(fill="blue") + coord_flip()
```



Sin embargo, ya existe en las respuestas una opción "otro", por lo una etiqueta más apropiada seria "Los demás" en vez de otros, para ello utilizamos la opción *other_leve* para personalizar el valor con el cual se agruparan las demás respuestas:

```
lapop %>%
mutate(problema_mas_grave = fct_lump(problema_mas_grave, n=10, other_level =
"Los demás")
     %>% fct_infreq() %>% fct_rev()) %>%
ggplot(aes(problema_mas_grave)) +
geom_bar(fill="red") + coord_flip()
```



5 Actividades

En la archivo *datos_imss.csv* se encuentra una muestra de la base de datos de trabajadores asegurados en el IMSS durante noviembre de 2020. Importa esta base de datos en un data frame llamado imss y calcula lo siguiente:

1. Convierte las variables *sexo, rango_edad y rango_salarial* a factores.

2. Consulta el archivo **descriptor_imss** en cual encontrarás el significado de las variables *sexo, rango_edad y rango_salarial*. Utiliza esta información para renombrar los factores por otros nombres más fáciles de entender. Por ejemplo, en vez de que una etiqueta sea *E2 que significa mayor a 1 y hasta 2 veces el salario mínimo puedes reemplazarla por 1-2 SM*

3. Gráfica el total de trabajadores por grupo de edad

4. Gráfica el total de trabajadores por rango de salario

Capítulo 11

Funciones

Capítulo 11 | Funciones

1 Introducción

En este capítulo trabajaremos con dos opciones que permitirán hacer mas eficiente la construcción de código. Profundizaremos en el uso de la familia de operadores del tipo *pipe* y aprenderemos a construir nuestras propias funciones. Los elementos que se trabajan en este capítulo harán que el proceso de programación sea mas elegante, sencillo e intuitivo. No sólo para nosotros, si no además para aquellos con los que tengamos que compartir el trabajo. Usaremos una base de datos, que contiene únicamente un conjunto de valores aleatorios, los cuales no tienen ninguna interpretación específica. Esto no es una limitante, pues las técnicas que aprendamos podrán ser replicadas a cualquier otra base de datos, de las diversas que hemos presentado en este curso.

2 Previos

Para este capítulo usaremos las librerías con las que hemos trabajado previamente. tibble la usaremos para construir otros *data frames* que nos permitan ver el funcionamiento de ciertos comandos. La librería **pryr** contiene un comando que nos ayudará a conocer el tamaño de una tabla de datos, mientras que **magrittr** nos permitirá profundizar en otros operadores de la familia *pipe*.

```
library(tibble)
library(tidyverse)
library(magrittr)
library(pryr)
library(readr)
```


3 Pipes con magrittr

En varios de los ejemplos que hemos desarrollado a lo largo de este curso, hemos utilizado el operador *pipe* que nos ayuda a concatenar un conjunto de operaciones sobre un objeto. Esta función se carga de manera automática cuando cargamos la librería *tidyverse* sin embargo, este operador forma parte de la librería *magritrr* la cual contiene otras herramientas y revisaremos en este capítulo.

Como ya te habrás dado cuenta el uso del operador *pipe* permite simplificar el código y nos ayuda a no tener que lidiar con una gran cantidad de objetos, que en ocasiones no son tan necesarios, pues solo surgen como pasos intermedios en un proceso.

Usemos la siguiente tabla de datos para ejemplificar las ventajas de *pipe*.

```
setwd("~/Dropbox/Curso de R/Cap11_Funciones")
dp←read_csv("Datos_Prueba.csv")
dp
```

```
## # A tibble: 15 x 4
##    a        b        c        d
##    <dbl>    <dbl>    <dbl>    <dbl>
## 1  -3.23    -1.77    1.62     -2.44
## 2  2.57     4.92     -1.66    2.28
## 3  3.52     1.52     -1.26    1.29
## 4  2.77     -8.45    -2.02    -1.49
## 5  4.80     2.44     0.0691   2.83
## 6  -0.718   -6.48    -0.395   -2.26
## 7  0.334    -0.274   4.17     -2.87
## 8  0.469    -5.99    2.50     -1.18
## 9  2.65     -6.40 -  5.87 -   0.858
## 10 1.25     6.38     3.59     -1.70
## 11 1.89     2.67     -0.805   -2.06
## 12 1.06     -5.28    0.454    -2.55
## 13 1.47     12.3     2.51     -0.243
## 14 4.49     8.87     -1.55    -1.28
## 15 -0.483   -7.45    -4.41    -5.55
```

Supón que a la tabla de datos anterior deseamos agregar una columna que sea igual a la multiplicación de **e=c*d**. Luego deseamos extraer solo los datos donde el valor de la nueva columna **e>0**.

Finalmente nos interesa que el resultado final contenga sólo los valores de la columna *a* que sean mayores al promedio de los datos contenidos en la columna *c*. Para construir la base de datos final, sin el uso del operador *pipe*, tendríamos que construir un conjunto de *data frames* intermedios, como los siguientes:

```
dp1<-mutate(dp, e=c*b)
dp2<-filter(dp1,e>0)
dp3<-filter(dp2,a>mean(dp2$c))
dpf<-select(dp3,a)
dpf
```

```
## # A tibble: 7 x 1
##    a
##    <dbl>
## 1  2.77
## 2  4.80
## 3  -0.718
## 4  2.65
## 5  1.25
## 6  1.47
## 7  -0.483
```

Cada uno de estos objetos ocupa un determinado espacio en la memoria de *R*.

```
object_size(dp)
```

```
## 3.79 kB
```

```
object_size(dp1)
```

```
## 4.06 kB
```

```
object_size(dp, dp1)
```

```
## 4.5 kB
```

El objeto **dp** requiere de 3.79 KB en memoria, mientras que **dp1** requiere 4.06. Juntos deberían requerir la suma de estos números, sin embargo, parece ser que juntos ocupan 4.5 KB. Esto sucede porque *R* sabe que el objeto **dp1** es una copia del objeto dp, a excepción de la columna e y no considera necesario duplicarlas, porque es algo que ambas contienen común. Esto hace que el uso de la memoria en *R* sea eficiente, aunque el código que hemos construido no lo es, pues ahora tenemos cinco objetos, y solo nos interesa el resultado final contenido en **df**. Por ahora eliminemos estos objetos que hemos creado de manera excesiva.


```
remove(dp1, dp2, dp3)
```

Pudimos haber disminuido la cantidad de objetos, si hubiéramos reemplazado el nombre del *data frame* varias veces, haciendo;

```
dp<-mutate(dp, e=c*b)
dp<-filter(dp,e>0)
dp<-filter(dp,a>mean(dp$c))
dp<-select(dp,a)
dp
```

```
## # A tibble: 7 x 1
##    a
##    <dbl>
## 1  2.77
## 2  4.80
## 3  -0.718
## 4  2.65
## 5  1.25
## 6  1.47
## 7  -0.483
```

Sin embargo, el abuso en el uso de estos reemplazos complica las revisiones y detección de errores, pues ahora ya no disponemos de la fuente original de datos, ya que en los reemplazos sucesivos ha sido modificada. Una opción mas elegante y práctica sería tener únicamente el *data frame* original y el final. El uso de *pipe* nos permite estas ventajas. Utilizando podemos hacer;

```
dp<-read_csv("Datos_Prueba.csv")
dpf<- dp %>%
     mutate(e=c*b) %>%
     filter(e>0) %>%
     filter(a>mean(c)) %>%
     select(a)
dpf
```

```
## # A tibble: 7 x 1
##    a
```

```
##    <dbl>
## 1  2.77
## 2  4.80
## 3  -0.718
## 4  2.65
## 5  1.25
## 6  1.47
## 7  -0.483
```

En este punto es importante una observación. En este caso hemos indicado que el procesamiento genere un objeto de nombre *dpf* en el que se asigne el resultado de todos los procesos efectuados. De no haber indicado **dpf<-** se hubiera efectuado el procesamiento de información sin almacenarla en un nuevo objeto y veríamos el resultado de esta manera.

```
dp %>%
     mutate(e=c*b) %>%
     filter(e>0) %>%
     filter(a>mean(c)) %>%
     select(a)
```

```
## # A tibble: 7 x 1
##    a
##    <dbl>
## 1  2.77
## 2  4.80
## 3  -0.718
## 4  2.65
## 5  1.25
## 6  1.47
## 7  -0.483
```

Nota que en ninguno de los dos casos se ha modificado la tabla de datos original. Entonces *pipe* concatena instrucciones **no** efectúa asignaciones *pipe* funciona gracias a que **R** usa *transformación léxica* reescribiendo los objetos intermedios, para mostrar únicamente el resultado final. Existe un tipo de aplicación en las que *pipe* no funcionará como esperamos. Esto sucede básicamente con funciones cuyos objetos, que ya existen en el *enviroment*. Por ejemplo, la función **assign()** crea una nueva variable con un determinado nombre del enviroment.


```
assign("t", 50)
t
```

```
## [1] 50
```

```
# Se asigna un valor de 50 a la variable t
```

```
"t" %>% assign(378)
t
```

```
## [1] 50
```

pipe ya no es capaz de asignar el nombre, pues la variable ya ha sido previamente definida en el **enviroment**. Para modificar esto debemos indicar que queremos modificar lo que hay en el *enviroment*

```
ambiente<-environment()
     "t" %>% assign(378, envir = ambiente)
t
```

```
## [1] 378
```

De esta manera creamos un ambiente donde deseamos que se encuentre la variable. Así podemos usar *assign* y *pipe* al mismo tiempo.

Observa también que si hacemos lo siguiente no se crea de manera explicita un objeto en el *enviroment* de nombre *h*. Aún así, entiende que debe concatenar dos cosas, de manera virtual un valor de 378 y a ese sumar 8.

```
"h" %>% assign(378) %>%
     sum(8)
```

```
## [1] 386
```

3.1 Cuando no usar pipes

Hemos visto que pipe genera muchas ventajas al momento de escribir código. Sin embargo, existen ciertas circunstancias en las cuales Hadley no recomienda su uso.

- No se recomienda no usar pipe cuando se tienen por decir mas de 10 instrucciones.
- Si se tienen múltiples entradas y múltiples salidas de un proceso. Es decir, se recomienda su uso cuando se trata de la manipulación de un mismo objeto que generará una única salida. Si se trata de dos objetos diferentes que deben ser combinados, no se recomienda el uso de *pipe*.

3.2 Otras herramientas de magrittr

Dentro de la paquetería **magrittr** existen tres funciones que alternativas que permiten la concatenación de instrucciones y son semejantes a *pipe*.

3.2.1 T pipe

Esta función **%T>%** se conoce como *T pipe* y trabaja de manera similar a *pipe*. La diferencia principal entre ellas radica en que *T pipe* regresa el lado izquierdo de la instrucción.

Observa el siguiente ejemplo en el que con el uso de la función **rnorm(20)** hemos generado de manera aleatoria un conjunto de 20 números, los cuales han sido distribuidos en dos columnas y acomodados en una matriz y hemos pedido que se calcule la suma de los números en cada columna. Al usar %>% el valor resultante en h es una matriz que contiene las sumas de cada columna (lo que esta a la derecha del último operador)

```
h<-rnorm(20) %>%
     matrix(ncol=2) %>%
     colSums
h
```

```
## [1] -2.5638120 0.8289139
```

Si vemos la clase de objeto que es h tenemos;

```
class(h)
```

```
## [1] "numeric"
```

En el siguiente código se replica el mismo procedimiento, pero en la segunda línea se usa **%T>%** por lo que la ejecución regresará, lo que se encuentra a la izquierda del operador, es decir la matriz con las dos columnas.

```
h<-rnorm(20) %>%
     matrix(ncol=2) %T>%
     colSums
h
```

```
##              [,1]          [,2]
## [1,]   0.03572779   -0.0006392553
## [2,]   1.16907222    0.5413506108
## [3,]  -1.46282795   -0.3053257980
## [4,]  -0.34794257   -0.0012483004
## [5,]  -0.48573196    1.3229983841
## [6,]  -0.85704831   -1.0172714615
## [7,]   1.50554251    0.8931270214
## [8,]  -1.03757856   -2.2872168999
## [9,]   0.01195682   -0.1609991564
## [10,] -0.75055188    0.1175504098
```


La clase de objeto ahora es;

```
class(h)
```

```
## [1] "matrix" "array"
```

3.2.2 Exposición pipe

Este operador expone los nombres de las variables en un data frame, para que sea posible referirse a ellos de manera explicita. Supongamos que del objeto datos de prueba **dp**, nos interesa la correlación (comando cor) que existe sólo entre las columnas a y b. Usamos **%$%** para poder referirnos a esas columnas de manera directa.

```
dp %$%
     cor(a, b)
```

```
## [1] 0.3680542
```

Observa que al ejecutar el siguiente código usando el operador *pipe* común obtendremos un error. Esto sucede porque aunque hemos indicado que estamos trabajando con **dp** la instrucción no es capaz de reconocer que a y b se refieren a los nombres de las columnas que existen en **dp**.

```
dp %>%
     cor(a, b)
# Error in cor(., a, b) : invalid 'use' argument
```

Observa que de otro modo, tendríamos que haber hecho;

```
cor(dp$a, dp$b)
```

```
## [1] 0.3680542
```

Lo que nos da la idea del funcionamiento de este operador %$%.

3.2.3 Asignación pipe

Este operador permite efectuar asignaciones y reemplazos en el código. Por ejemplo, usemos los datos de prueba contenidos en dp. Comparemos *pipe* con *Asignación pipe*

```
dp %>% mutate(e=a+b)
```

```
## # A tibble: 15 x 5
##         a      b      c      d      e
##         <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1       -3.23  -1.77  1.62   -2.44  -5.00
## 2       2.57   4.92   -1.66  2.28   7.49
## 3       3.52   1.52   -1.26  1.29   5.04
## 4       2.77   -8.45  -2.02  -1.49  -5.68
## 5       4.80   2.44   0.0691 2.83   7.23
## 6       -0.718 -6.48  -0.395 -2.26  -7.20
## 7       0.334  -0.274 4.17   -2.87  0.0594
## 8       0.469  -5.99  2.50   -1.18  -5.52
## 9       2.65   -6.40  -5.87  -0.858 -3.74
## 10      1.25   6.38   3.59   -1.70  7.63
## 11      1.89   2.67   -0.805 -2.06  4.57
## 12      1.06   -5.28  0.454  -2.55  -4.22
## 13      1.47   12.3   2.51   -0.243 13.8
## 14      4.49   8.87   -1.55  -1.28  13.4
## 15      -0.483 -7.45  -4.41  -5.55  -7.93
```

Si observamos se efectúa el procedimiento, pero no hay una asignación. Es decir, no se modifica la tabla de datos original. Podemos ver los primeros valores de **dp** y darnos cuenta qué sólo hay cuatro columnas, pues la columna e no existe.

```
head(dp)
```

```
## # A tibble: 6 x 4
##   a      b      c      d
##   <dbl>  <dbl>  <dbl>  <dbl>
## 1 -3.23  -1.77  1.62   -2.44
## 2 2.57   4.92   -1.66  2.28
## 3 3.52   1.52   -1.26  1.29
## 4 2.77   -8.45  -2.02  -1.49
## 5 4.80   2.44   0.0691 2.83
## 6 -0.718 -6.48  -0.395 -2.26
```


En cambio, si usamos asignación pipe

```
dp %<>% mutate(e=a+b)
head(dp)
```

```
## # A tibble: 6 x 5
##   a      b      c      d      e
##   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 -3.23  -1.77  1.62   -2.44  -5.00
## 2 2.57   4.92   -1.66  2.28   7.49
## 3 3.52   1.52   -1.26  1.29   5.04
## 4 2.77   -8.45  -2.02  -1.49  -5.68
## 5 4.80   2.44   0.0691 2.83   7.23
## 6 -0.718 -6.48  -0.395 -2.26  -7.20
```

Observa que ahora en **dp,** si se ha efectuado una transformación de los datos, lo cual no sucede en el primer caso.

4 Funciones

Cuando nos encontramos en casos reales, trabajando con bases de datos y códigos que requieren grandes cantidades de programación es común que en ocasiones una misma actividad se repita mas de una vez. Considera por ejemplo que tienes tres rectángulos, y que necesitas obtener el área de cada uno de ellos. Una forma de hacerlo es declarar, todas las variables necesarias y calcular tres veces el área usando la definición conocida como la multiplicación de su base por su altura. Ese procedimiento es el que usamos a continuación;

```
#Rectángulo 1
base1<-40
altura1<-50
```

```
base1*altura1
```

```
## [1] 2000
```

```
#Rectángulo 2
base2<-200
altura2<-60
base2*altura2
```

```
## [1] 12000
```

```
#Rectángulo 1
base3<-4
altura3<-150
base3*altura3
```

```
## [1] 600
```

Observa que en este caso hemos tenido que escribir prácticamente lo mismo tres veces. Para ello una buena idea es copiar y pegar el código y sólo cambiar los elementos que necesitamos 1, por 2 y 2 por 3. Si embargo, esto incrementa el tamaño y puede llevarnos a cometer una mayor cantidad de errores en el proceso. Una forma de solucionar esto es hacer uso de funciones. Una función nos permite automatizar determinadas tareas de una forma mas sofisticada que sólo copiar y pegar. Escribir una función nos proporciona tres ventajas importantes;

1. Mediante la asignación de nombres relacionados a la actividad que realiza la una función, podemos hacer nuestro código mas simple de entender.
2. Si los requerimientos cambian, no es necesario modificar todas las veces que se copió y pegó el código. Será necesario únicamente modificar la función.
3. Elimina el riesgo de cometer errores al copiar y pegar código.

Por ejemplo, retomando el caso de calcular el área de tres rectángulos, podemos hacer una función que se llamada **area** que nos otorga el mismo funcionamiento, pero con mayor facilidad. Para crear una función usamos el comando **function** y especificamos entre paréntesis, nombres genéricos a las variables que requiere la función para su uso. En este caso usamos dos, pues se requiere una variable para la base y otra para la altura.

```
area<-function(x,y){
     x*y
}
area(40,50)
```

```
## [1] 2000
```



```
area(200,60)
```

```
## [1] 12000
```

```
area(4,150)
```

```
## [1] 600
```

Observa que una vez que hemos nombrado a la función lo siguiente es sólo llamarla y asignarle valores a las variables que consume de entrada. De está manera al escribir **area(40,50) R** entiende que **x=40, y=50** y que el procesamiento que debe hacer es multiplicarlas.

Para construir una función se requieren tres componentes;

1. El nombre de la función. En este caso la recomendación es que el nombre se relacione con el proceso que realiza la función.
2. Las variables de entrada o argumentos que consume la función. Podemos hacer uso de letras como (w,x,y,z)
3. El código con las instrucciones sobre como debe operar la función se conoce como el **cuerpo** de la *función* y si se trata de una cantidad de código, se encuentra delimitado por llaves de inicio y fin {}. En ocasiones si se trata de pocas lineas podemos omitir las llaves. Sin embargo, su uso es ampliamente recomendado en cualquier caso.

Consideremos un ejemplo un tanto mas complejo. Supongamos que tenemos el siguiente *data frame*, que hemos construido usando la función **tibble()**. La función **rlnorm(5)** indica que se generen 5 valores aleatorios.

```
df <- tibble(
a = rlnorm(5),
b = rlnorm(5),
c = rlnorm(5),
d = rlnorm(5)
)
df
```

```
## # A tibble: 5 x 4
##    a       b       c       d
##    <dbl>   <dbl>   <dbl>   <dbl>
## 1  0.302   1.04    0.435   3.68
## 2  2.01    0.303   0.0808  0.159
## 3  1.20    1.59    0.712   1.67
## 4  4.65    0.230   4.39    0.214
## 5  0.482   0.722   2.67    3.24
```

Este *data frame* tiene diferentes valores. Supón que nos interesa convertir cada valor en un número de 0 a 1, de manera que los números obtenidos este re-escalados. Es decir que el valor mas pequeño de cada columna sea 0, mientras que el mas grande sea 1 y así sucesivamente. Usando la función **range()** que genera valor máximo y mínimo, tenemos que para la primera columna tendríamos que hacer;

```
rango<-range(df$a, na.rm = TRUE)
df$a<-(df$a-rango[1])/(rango[2]-rango[1])
df
```

```
## # A tibble: 5 x 4
##    a         b         c          d
##    <dbl>     <dbl>     <dbl>      <dbl>
## 1  0         1.04      0.435      3.68
## 2  0.393     0.303     0.0808     0.159
## 3  0.207     1.59      0.712      1.67
## 4  1         0.230     4.39       0.214
## 5  0.0414    0.722     2.67       3.24
```

Observa que dentro de **range(df$a, na.rm = TRUE)** indicamos *na.rm = TRUE* para asegurarnos que solo se efectúe esta operación con valores diferentes a *NA*.

La función **range()** genera el máximo y el mínimo. Cada uno se registra dentro de la variable rango. En la posición uno está el mínimo en la dos el máximo. De esta manera los valores de la columna a han sido re-escalados. Para completar el proceso, tendríamos que hacer esto tres veces mas. Una opción sería copiar y pegar cambiando en cada caso, a por b, luego por c y así sucesivamente.

```
rangob<-range(df$b, na.rm = TRUE)
df$b<-(df$b-rangob[1])/(rangob[2]-rangob[1])
rangoc<-range(df$c, na.rm = TRUE)
df$c<-(df$c-rangoc[1])/(rangoc[2]-rangoc[1])
rangod<-range(df$d, na.rm = TRUE)
df$d<-(df$d-rangod[1])/(rangod[2]-rangod[1])
```

Esta opción requiere ser extremadamente cuidadosos, pues podemos cometer errores en el reemplazo de *a* por *b*

Otra opción es hacer una función;

```
re_escala<-function(x){
rango<-range(x, na.rm = TRUE)
```



```
(x-rango[1])/(rango[2]-rango[1])
}
```

De esta manera, solo asignamos a cada columna, la función que hemos creado;

```
df$b<-re_escala(df$b)
df$c<-re_escala(df$c)
df$d<-re_escala(df$d)
df
```

```
## # A tibble: 5 x 4
##    a         b         c          d
##    <dbl>     <dbl>     <dbl>      <dbl>
## 1  0         0.597     0.0820     1
## 2  0.393     0.0535    0          0
## 3  0.207     1         0.146      0.430
## 4  1         0         1          0.0156
## 5  0.0414    0.361     0.599      0.877
```

Observa que la función que llamamos re_escala tiene una única entrada, que hemos representado en la función con la letra x.

4.1 Recomendaciones

Trabajar con funciones genera una ventaja importante en el manejo de grandes líneas de código. En estos casos la recomendación es que los nombres que asignemos a las funciones sean claros, cortos y lo mas importante; que puedan proporcionar algo de información sobre la funcionalidad.

Se recomienda el uso de _, así como combinaciones de mayúsculas y minúsculas, pero procura siempre guardar consistencia. Dentro del cuerpo de la función es posible el uso de comentarios con el símbolo # por lo que pueden incluirse descripciones. Sobra decir que **aunque R te lo permita** nunca uses un nombre reservado de **R** para una función que tu construyas. Mas delante vemos un ejemplo de esto.

Siempre que escribas código recuerda que debes ser lo mas claro posible, ya sea que compartas ese código con alguien mas o que tú mismo necesites revisarlo mas adelante.

4.2 Una condicional

En ocasiones es necesario que se ejecute una parte de una función en determinada circunstancia y otras partes bajo circunstancias distintas. Para ello hacemos uso de condicionales. La estructura básica de una condicional es;

```
if(condicion){
# que deseamos cuando la condición se verdadera
} else {
# que deseamos cuando la condición sea false
}
```

Por ejemplo, considera una función que deseamos que nos indique con texto si un número es positivo o negativo. Para ello hacemos;

```
signo<-function(x){
   if (x>0) {
      print("Positivo")
   } else {
      print("Negativo")
   }
}
```

Observa que la condicional indica que, si el número es mayor de cero, se regresa la palabra Positivo. Podemos probar esta función;

```
signo(4)
```

```
## [1] "Positivo"
```

```
signo(-34)
```

```
## [1] "Negativo"
```

Al indagar que regresa la función bajo el número 0, nos daremos cuenta que, regresa *Negativo*, pues el cero no es mayor que el mismo.

La *condición* dentro del *if* siempre debe ser una condición que sea necesario evaluar, y regresar un valor FALSO o VERDADERO, dependiendo de su cumplimiento. Al establecer la condición podemos hacer uso de los caracteres;

- && (y) para indicar que deseamos que se cumplan dos condiciones a la vez
- **||** (o) para indicar que deseamos que se cumpla una u otra condición

Nunca uses **&** o **|** en una condicional *if* las cuales están reservadas para vectores de datos, no para una sola condición. Recuerda siempre que la condición establecida en *if* debe regresar un único valor.

4.3 Múltiples condicionales

Es posible escribir mas de una condicional en una función. Para ello usamos;

```
if(condicion){
# haz esto
} else if (otra condicion) {
# haz lo otro
} else {
# otra instrucción
}
```

Considera el ejemplo de función **signo** donde deseamos que en caso de que se trate del cero, no indique ni positivo ni negativo.

```
signo<-function(x){
  if (x>0) {
     print("Positivo")
  } else if (x==0) {
     print("No lo sé")
  } else {
     print("Negativo")
  }
}
```

Probamos la función

```
signo(-133)
```

```
## [1] "Negativo"
```

```
signo(5)
```

```
## [1] "Positivo"
```

```
signo(0)
```

```
## [1] "No lo sé"
```

En caso de que la función requiera una gran cantidad de condicionales, podemos usar la opción **switch** y **stop** en lugar de escribir todas las condiciones. Por ejemplo, considera una función que permita calcular alguna de las operaciones básicas entre dos números;

```
operacion<-function(x,y,opera){
     switch(opera,
          "suma"=x+y,
          "resta"=x-y,
          "multiplica"=x*y,
          "divide"=x/y,
          stop("Operación desconocida"))
}
```

Observa que la función consume tres entradas o argumentos, los números y la operación. Podemos probarla haciendo;

```
operacion(4,8,"suma")
## [1] 12
```

```
operacion(4,8,"resta")
## [1] -4
```

```
operacion(4,8,"multiplica")
## [1] 32
```

```
operacion(4,8,"divide")
## [1] 0.5
```

Prueba haciendo **operacion(4,8,"potencia")** ¿Qué obtienes? ¿Porqué?. En este caso obtendremos un error de operación desconocida.

Tanto la función *if* como function van acompañadas de un par de llaves, que marcan el inicio y el final de las instrucciones en ella contenidas. En este caso la recomendación es que uses cambios entre diferentes líneas para que el código que desarrolles seas mas legible.

4.4 Argumentos de una función

Las funciones que hemos construido hasta ahora usan un sólo tipo de argumento; los datos para hacer el cálculo requerido. Una función puede contener dos tipos de argumentos; *datos* (como los ejemplos que hemos efectuado) y *detalles* los cuales controlan ciertos detalles del cálculo efectuado. Por ejemplo, al consultar la ayuda de la función **mean()** tenemos

```
mean(x, ...)
```



```
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Esto significa que además del conjunto de datos se requiere un valor para **trim** y para **na.rm**. En caso de que estos *detalles* no se proporcionen por parte del usuario, la función, de forma automática toma los valores marcados. De esta manera no genera errores en la lectura y ejecución. Algo semejante sucede con la función **str_c**.

Para construir una función con algún argumento de tipo *detalles*, es necesario declarar la entrada con su respectivo valor.

Por ejemplo, construyamos una función de nombre **veces** que de manera automática regrese un número multiplicado por dos, pero que permita la opción de cambiar las veces por las que el número se deba multiplicar, según la indicación del usuario.

```
veces<- function(x, num=2) {
     num*x
}
```

Si ponemos a prueba la función

```
veces(2)
## [1] 4
```

```
veces(2,num=5)
## [1] 10
```

Observa que la función veces, tiene un valor predefinido de **num=2** por lo cual si el usuario no proporciona ese valor no hay problema. Caso contrario si hacemos la función

```
veces2<- function(x, num) {
     num*x
}
```

Y la probamos

```
veces2(2)
#Error in veces2(2) : argument "num" is missing, with no default
```

Obtenemos un resultado como error, pues para poder completar su funcionamiento la función requiere dos argumentos y solo hemos proporcionado uno.

Siempre que ejecutemos una función y deseemos cambiar el valor default de uno de su argumentos tipo detalle debemos incluir su etiqueta, con el fin de evitar confusiones. Si hay más de un argumento en una función, deben ir separados por comas (,) y luego de cada coma debemos asegurar un espacio.

En ocasiones cuando usamos funciones es necesario asegurarnos que ciertas condiciones se cumplen. Por ejemplo, si deseamos efectuar el producto punto de dos vectores.

```
punto<-function(x,y){
      x*y
}
a<-1:3
a
```

```
## [1] 1 2 3
```

```
b<-1:6
b
```

```
## [1] 1 2 3 4 5 6
```

```
punto(a,b)
```

```
## [1] 1 4 9 4 10 18
```

Esto constituye un error ya que el producto punto de dos vectores, debe regresar la multiplicación de cada entrada del vector, por lo que debe tratarse de vectores del mismo tamaño.

En este caso primero debemos asegurarnos que los vectores tienen la misma longitud y en caso de que no sea así, alertar al usuario de la función sobre la existencia de un error. Lo correcto en este caso es;

```
punto2<-function(x,y){
  if (length(x)!=length(y)){
      stop( "`x`, `y` deben ser del mismo tamaño")
  } else {
      x*y
  }
}
```

Probando la nueva función;


```
punto2(a,b)
#Error in punto2(a, b) : `x`, `y` deben ser del mismo tamaño
```

Es posible también el uso de la opción stopifnot que indica que la función parará en caso de que no se cumpla la condición

```
punto3<-function(x,y){
      stopifnot(length(x)==length(y))
      x*y
}
```

Al ejecutarla el resultado nos indica que la condición que establecimos para que la función continúe no se cumple

```
punto3(a,b)
#Error in punto3(a, b) : length(x) == length(y) is not TRUE
```

4.4.1 Múltiples entradas (. . .)

Algunas funciones para su operación requieren una gran entrada de argumentos. Considera por ejemplo que queremos una función que multiplique por 10 la media de un conjunto de datos. Nos resulta imposible escribir todos los argumentos que puede tener la función, así que usamos ... para referirnos a todos ellos.

```
multiple<-function(...){
      10*mean(...)
}
multiple(1:10)
```

```
## [1] 55
```

4.5 Valores de regreso

Los valores de regresos son aquellos para los cuales la función ha sido creada. Por ejemplo, la función que hemos construido a la que nombramos **multiple** regresa 10 veces la media de una conjunto de datos que se introducen.

En las funciones que hemos construido, se ha obviado el especificar el valor de regreso, pues se este se encuentra hasta el final del cuerpo de la función. En caso de que sea necesario que la función regrese un valor antes del final, podemos usar el comando **return().**

```
complejo<-function(x,y){
  if(length(x)==0||length(y)==0){
     return(0)
  } else {
     x+y
  }
}
complejo(1,5)
```

```
## [1] 6
```

```
complejo(NULL,1)
```

```
## [1] 0
```

De esta manera hacemos hecho explicito el resultado de la función que estamos construyendo.

Pareciera que no resulta tanto importante hablar sobre los valores que regresa una función. Sin embargo, cuando construimos funciones conocidas como *pipeables* (piensa en ello como el operador pipe %>%) hablar del resultados cobra relevancia. Existen dos tipos de funciones; transformación y efecto. El primer tipo son funciones que realizan la trasformación de un objeto, mientras que el segundo son solo funciones que realizan la operación con la información contenida en el objeto.

```
#Efecto
mean(df$a)
```

```
## [1] 0.3282897
```

```
#Transformación
df %>%
  mutate(e=d*c)
```

```
## # A tibble: 5 x 5
##    a         b         c         d         e
##    <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1  0         0.597     0.0820    1         0.0820
## 2  0.393     0.0535    0         0         0
## 3  0.207     1         0.146     0.430     0.0629
## 4  1         0         1         0.0156    0.0156
## 5  0.0414    0.361     0.599     0.877     0.526
```

En el ejemplo anterior, la primera función es de *efecto* mientras que la segunda es de *transformación*.

Considera que deseamos construir una función que contabilice el número de valores mayores a 0.6 en un *data frame*.

```
mayores<-function(x){
     n<-sum(x>0.6)
     cat("Hay", n, "valores mayores a 0.6 \n")
     invisible(x)
}
```

Parece que en esta función únicamente tenemos como resultado el número de valores mayores a 0.6. Sin embargo, al indicarle **invisible(x)** le estamos diciendo que también considere **x** aunque no la muestre.

Asignemos a un nuevo objeto en **R** el resultado de la función que construimos de nombre *mayores*

```
datos<-mayores(df)
```

```
## Hay 5 valores mayores a 0.6
```

Veamos que clase de objeto se trata:

```
class(datos)
```

```
## [1] "tbl_df" "tbl" "data.frame"
```

```
dim(datos)
```

```
## [1] 5 4
```

Observa que es un objeto de tipo *data frame,* pues hemos construido la función para que el data frame que consume este contenido en su resultado de forma invisible. Para comprobar este punto, construyamos la misma funciones sin incluir **invisible(x)** y veamos la clase de objeto que se trata.

```
mayores2<-function(x){
     n<-sum(x>0.6)
     cat("Hay", n, "valores mayores a 0.6 \n")
}
datos2<-mayores2(df)
```

```
## Hay 5 valores mayores a 0.6
```

Observa que ahora que datos2 regresa un valor nulo.

Construir funciones como mayores haciendo uso del comando invisible permite que la funciones que construyamos se puedan operar con pipe ya que al contener en este caso el data frame de origen, permite seguir operándolo.

Veamos esto usando la función mayores en combinación con el operador pipe.

```
df %>%
mayores() %>%
mutate(e=c*d) %>%
mayores()
```

```
## Hay 5 valores mayores a 0.6
```

```
## Hay 5 valores mayores a 0.6
```

Si intentas hacer el mismo procedimiento usando **mayores2** te darás cuenta que **R** regresa un error, pues no cuenta con materia prima (en este caso un *data frame*), para continuar con la operación.

4.6 Enviroment

Si recuerdas en los primeros capítulos de este curso hablamos de la venta *enviroment* donde se muestran los diferentes objetos con los que *R* trabaja durante una sesión de trabajo. En el desarrollo de este capítulo, hasta el momento, tu *enviroment* luce mas o menos así:

Environment | History | Connections | Tutorial

Import Dataset | Grid

Global Environment

Name	Type	Len...	Size	Value
a	integ...	3	0 B	int [1:3] 1 2 3
altura1	numer...	1	56 B	50
altura2	numer...	1	56 B	60
altura3	numer...	1	56 B	150
ambiente	envir...	5	56 B	Environment
area	funct...	1	4.7 ...	function (x,...
b	integ...	6	0 B	int [1:6] 1 2 ...
base1	numer...	1	56 B	40

Si observas con cuidado podrás ver que cada objeto que hemos creado se refleja en ese espacio y se incluye el tipo de objeto de que se trata. Puedes ver las funciones que hemos construido. Siempre que construyamos una función debemos asegurarnos que se encuentra en el *enviroment* para que pueda ejecutarse adecuadamente. Siempre que pedimos a **R** que opere de alguna manera un objeto, lo buscará primero en el *enviroment.*

Considera por ejemplo la siguiente función


```
prueba_error<-function(x){
      x*a
}
prueba_error(5)
```

```
## [1] 5 10 15
```

Dentro de otros lenguajes de programación, esto sería un error, pues la función consume una variable de nombre a que no ha sido proporcionada como un argumento. Para **R** esto no es un error, pues rápidamente identifica que en tu *enviroment* hay un objeto de nombre a quieres operarlo dentro de la función.

Si eliminamos el objeto a previamente construido y ejecutamos nuevamente, tenemos;

```
remove(a)
prueba_error<-function(x){
      x*a
}
prueba_error(5)
#Error in prueba_error(5) : object 'a' not found
```

Esto sucede ya que **R** opera con reglas llamadas de alcance léxico lo que implica que siempre que se pueda construir un objeto con determinado nombre y tal objeto exista en el *enviroment*, puede ser usado. Esta podría ser una desventaja, por ello procura siempre que las variables que uses dentro de la función tengan sentido en relación a los argumentos.

Este comportamiento de alcance léxico también permite construir función con nombres como esta;

```
`+`<-function(x,y) {
      x*y
}
5+8
```

```
## [1] 40
```

¿5+8=40? ¿Cómo es esto posible? Esto sucede porque hemos construido una función de nombre + que hace algo diferente a la suma. Si no somos los suficientemente cuidadosos con el uso, podemos tener muchos problemas con el manejo de *R*. Por lo pronto para evitar errores, borremos esta función

```
remove(`+`)
5+8
```

```
## [1] 13
```

5 Actividades

1. Modifica el siguiente código para que se obtenga exactamente el mismo resultado (una gráfica), pero con el uso de los operadores de la familia *pipe*. Para ello necesitarás la base de datos *Datos Prueba*. ¿Que ventaja observas en el uso de *pipe*?

```
dp1<-select(dp, c,d)
dp2<-mutate(dp1, e=c/d)
dp3<-filter(dp2,e>0)
dp4<-select(dp3, e)
plot(dp4)
```

2. ¿Cómo podrías hacer mas eficiente el siguiente código?

```
parte_1<-mutate(dp, h=c*d)
cor(parte_1$h, parte_1$d)
```

```
## [1] -0.4109446
```

3. Construye una función que pueda calcular la media ponderada de un conjunto de datos. Puedes documentarte sobre el concepto aquí Media_Ponderada. Usa el ejemplo 1 que ahí se proporciona para verificar que tu función fue construida correctamente.

4. Construye una función de nombre "saludo" la cual cuando se ejecute regrese, "Qué tengas un buen" y complemente con el día de la semana.

5. Construye una función pipeable que pueda indicar si un número es par o impar. Usa el operador Módulo **%%**

Capítulo 12

Vectores

Capítulo 12 | Vectores

1 Introducción

Un vector se puede considerar como un arreglo, el cual puede tener n filas con una columna. Podemos llamar a esto un vector columna. Aunque también podemos tener un arreglo de una fila con n columnas, el cual es un vector fila. Los vectores son los objetos que subyacen los *data frame*. Los vectores en **R** se refieren a un arreglo de tipo columna. Los elementos que componen un vector pueden ser numéricos o de texto.

En este capítulo utilizaremos algunas funciones de la paquetería *purrr*, la cual esta incluida en *tidyverse*. En esta ocasión no necesitamos carga una base de datos.

Recordemos iniciar nuestro script de la manera ya acostumbrada

```
library (tidyverse)
```

2 Vectores

En R existen dos tipos de vectores:

- Vectores atómicos. Estos pueden ser de diferentes tipos según el tipo de variable que contenga
  - lógico
  - entero
  - doble
  - carácter

  - completo
  - sin procesar (raw)
- Lista. Las cuales a veces son llamadas vectores recursivos debido a que pueden contener otras listas. Es como tener un vector dentro de otro vector.

La diferencia principal entre ambos tipos de vectores es que los atómicos son vectores homogéneos (todos los elementos que contiene son del mismo tipo), mientras que las listas pueden ser heterogéneas (contienen elementos de diferente tipo).

Cada vector tiene dos propiedades clave, su tipo y longitud. La instrucción *typeof()* nos indica el tipo de vector de que se trata, mientras que la instrucción *lenght()* determina la longitud del vector:

Usemos por ejemplo **letters**, el cual consiste en conjunto de datos almacenados de manera automática en *R*.

```
letters
```

```
##[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

Veamos de que tipo de vector se trata

```
typeof(letters)
```

```
## [1] "character"
```

Construyamos otros vectores (atómicos y listas) y veamos de que tipo son;

```
#Vectores atómicos
typeof("Hola mundo") # Un vector con una cadena de texto
```

```
## [1] "character"
```

```
typeof(1:10) #Vector con los números del 1 al 10
```

```
## [1] "integer"
```

```
typeof(0.1) # Un vector con el número 0.1
```

```
## [1] "double"
```

```
x ← seq(1:100) # Un vector con una sucesión de números del 1 al 100
length(x)
```

```
## [1] 100
```

```
# Listas
y <- list("a", "b", 1:100) # Una lista con tres elementos
typeof(y)
```

```
## [1] "list"
```

```
length(y)
```

```
## [1] 3
```

```
z <- list("Juan", "Maria", "Carmen") # Una lista con tres elementos
typeof(z)
```

```
## [1] "list"
```

```
length(z)
```

```
## [1] 3
```

Nótese que y es una lista de longitud 3, sin embargo, el tercer elemento es a su vez una lista de 100 elementos. Exploremos un poco la lista *y*.

```
typeof(y)
```

```
## [1] "list"
```

```
y
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1]     1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
## [19]    19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37]    37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
## [55]    55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
```



```
## [73]    73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
## [91]    91 92 93 94 95 96 97 98 99 100
```

Observa la estructura del resultado de ejecutar y. En este caso vemos que el resultado se muestra en tres partes la primera parte solo muestra *a*, la segunda *b*, mientras que la tercera, todos los números del 1 al 100.

Si queremos acceder solo al tercer elemento de la lista, debemos hacer

```
y[[3]]
```

```
## [1]   1  2  3  4  5  6  7  8  9   10 11 12 13 14 15 16 17 18
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
## [55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
## [73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
## [91] 91 92 93 94 95 96 97 98 99 100
```

```
length(y[[3]])
```

```
## [1] 100
```

O por ejemplo al primer elemento de la lista *y*

```
y[[1]]
```

```
## [1] "a"
```

```
y[[3]]
```

```
## [1]   1  2  3  4  5  6  7  8  9   10 11 12 13 14 15 16 17 18
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
## [55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
## [73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
## [91] 91 92 93 94 95 96 97 98 99 100
```

Exploremos ahora un poco el vector atómico *x*

```
typeof(x)
```

```
## [1] "integer"
```

```
x
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
## [55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
## [73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
## [91] 91 92 93 94 95 96 97 98 99 100
```

```
x[2]
```

```
## [1] 2
```

Para el caso de la lista *z*

```
z
```

```
## [[1]]
## [1] "Juan"
##
## [[2]]
## [1] "Maria"
##
## [[3]]
## [1] "Carmen"
z[[2]]
## [1] "Maria"
```

Observa que para acceder a los elementos del vector *x* se indica entre corchetes *[ ]* y la fila del elemento que nos interesa. En cambio, para acceder a los elementos de una lista es necesario utilizar dos corchetes *[[ ]]*. Debe quedar claro que aunque las listas también funcionan si utilizamos solo una vez los corchetes, pueden surgir errores, sobre todo si deseamos utilizar un elemento de una lista recursiva, es decir, una lista que esta dentro de una lista. Observa el siguiente ejemplo utilizando la lista que creamos anteriormente de nombre *y:*


```
# Obtener el elemento 1 de la lista y
y[1] #Forma incorrecta pero resultado correcto
```

```
## [[1]]
## [1] "a"
```

```
y[[1]] #Forma correcta
```

```
## [1] "a"
```

```
# Obtener el elemento 2 de la lista y
y[2] #Forma incorrecta pero resultado correcto
```

```
## [[1]]
## [1] "b"
```

```
y[[2]] #Forma correcta
```

```
## [1] "b"
```

```
# Obtener  la lista del elemento 3 de la lista y
y[3] #Forma incorrecta pero resultado correcto
```

```
## [[1]]
## [1]  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
## [55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
## [73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
## [91] 91 92 93 94 95 96 97 98 99 100
```

```
y[[3]] #Forma correcta
```

```
## [1]  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
## [55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
## [73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
## [91] 91 92 93 94 95 96 97 98 99 100
```

```
# Obtener el primer elemento de la lista que forma el elemento 3 de y
```

```
y[3][1] #Forma incorrecta y resultado incorrecto
```

```
## [[1]]
## [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
## [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
## [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
## [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
## [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
## [91]  91  92  93  94  95  96  97  98  99 100
```

```
y[[3]][1] #Forma correcta
```

```
## [1] 1
```

Observa que en este caso nos interesa sólo un número, y al ejecutar **y[3][1]** un conjunto de números pues *R* no es capaz de diferenciar esta instrucción de **y[3].**

```
# Obtener el último elemento de la lista que forma el elemento 3 de y
y[[3]][length(y[[3]])]
```

```
## [1] 100
```

Resulta útil entender estas diferencias, ya que algunas funciones guardan valores en listas o *data frames* y si deseamos recuperar un valor en especifico necesitamos conocer como poder acceder a estos elementos.

3 Vectores atómicos

3.1 Vectores lógicos

Los vectores del tipo lógico pueden tomar dos valores, verdadero (TRUE) o falso (FALSE), o en su caso valor perdido (NA). Los vectores lógicos son construidos con operadores de comparación.

- TRUE, o verdadero, cuando se cumple una condición
- FALSE, o falsa, cuando no se cumple una condición

Por ejemplo, en el siguiente código estamos creando el vector de nombre *x* con una serie de valores numéricos. Después de creado el vector *x* creamos un vector *y* que contendrá verdadero si los números contenidos en *x* cumple la condición mayor a 100 y menor a 1000 x > 100 & x<1000

```
x ← c(10,20,40,80,90,200,500,600,700,800,1000)
```



```
x
```

```
## [1] 10 20 40 80 90 200 500 600 700 800 1000
```

```
typeof(x)
```

```
## [1] "double"
```

```
y ← (x > 100 & x<1000)
y
```

```
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE
```

3.2 Vectores númericos

Puede ser un número entero o tipo doble. En **R**, de manera predeterminada los números son presentados como *double* y para generar un número entero, solo debemos colocar una *L* después de número:

```
typeof(1)
```

```
## [1] "double"
```

```
typeof(1L)
```

```
## [1] "integer"
```

Los números doble son aproximaciones. Representan números de punto flotante, por lo que no siempre pueden ser representados de manera precisa.

3.3 Vector carácter

Los vectores de carácter o texto son los tipos de vectores más complejos, ya que cada elemento del mismo es una cadena de caracteres y cada elemento puede tener una cantidad arbitraria de caracteres. Una característica importante de una cadena de caracteres en **R**, es que el software usa una reserva global de texto. Esto significa que cada texto (cadena de caracteres) único solo es almacenado en la memoria una vez y cada uso de ese texto apunta a esa representación.

Ahora aprenderemos algunas herramientas para trabajar con este tipo de vectores.

3.4 Coerción

Para realizar una conversión de un tipo de vector a otro (forzar o coercionar) existen dos métodos;

1. La coerción explícita ocurre cuando llamas a una función como *as.logical(), as.integer(), as.double()* o *as.character().* En estos casos el usuario asigna de el tipo de vector deseado según sus necesidades.

2. La coerción implícita ocurre cuando usas un vector en un contexto específico en el que se espera que sea de cierto tipo. Un ejemplo de ello es si tenemos un vector lógico (TRUE, FALSE) y le aplicamos una función numérica, como mean, entonces se asigna el valor de 1 a TRUE y de 0 a FALSE

Del ejamplo anterior tenemos;

```
x <- c(10,20,40,80,90,200,500,600,700,800,1000)
y <- x > 100 & x<1000
sum(y)
```

```
## [1] 5
```

```
mean(y)
```

```
## [1] 0.4545455
```

Un vector atómico no puede contener una mezcla de diferentes tipos, ya que el tipo es una propiedad del vector completo y no de los elementos individuales. Recuerda que esta es la diferencia fundamental de estos, con las listas. Si se necesita mezclar diferentes tipos en el mismo vector entonces se debe utilizar una lista.

3.5 Tipo de vector

En ocasiones nos interesa realizar tareas basadas en tipo de vector. Ya mostramos que podemos utilizar la función *typeof()*. Sin embargo, podemos utilizar una función alternativa para identificar si nuestro vector de interés es de un tipo en especifico, obteniendo por resultado un valor lógico (verdadero o falso). Para esto utilizamos la función *is.x*, donde *x* representa los diferentes tipos de vector:

```
# Retomando y del ejemplo anterior:
is.logical(y)
```

```
## [1] TRUE
```

```
is.integer(y)
```

```
## [1] FALSE
```

```
is.double(y)
```

```
## [1] FALSE
```

```
is.numeric(y)
```

```
## [1] FALSE
```



```
is.character(y)
```

```
## [1] FALSE
```

```
is.atomic(y)
```

```
## [1] TRUE
```

```
is.list(y)
```

```
## [1] FALSE
```

```
is.vector(y)
```

```
## [1] TRUE
```

También podemos utilizar esta forma para identificar el tipo de una escalar:

```
is_scalar_atomic(1)
```

```
## [1] TRUE
```

```
is_scalar_logical("A")
```

```
## [1] FALSE
```

```
is_scalar_character("A")
```

```
## [1] TRUE
```

Un escalar se refiere a una vector de una sola fila. Es decir, un vector compuesto por un único elemento.

3.6 Escalares

Así como R coerciona implícitamente los vectores para ser compatibles, también coerciona la longitud de los vectores, a esto se le llama reciclado de vectores, debido a que el vector de menor longitud se repite, o reciclan hasta igualar la longitud del vector más largo.

Esto es más útil cuando se trabaja con vectores y escalares. La mayoría de las funciones en **R** están vectorizadas, lo que implica que operan con un vector de números y es la razón por la cual las operaciones matemáticas no necesitan un interacción explicita cuando se realizan cálculos sencillos.

Es sencillo inferir que pasa sí sumamos dos vectores de la misma longitud, o un vector con un escalar:

```
#Suma de dos vectores de misma longitud
a <- c(1,1,1,1,1,1)
a2 <- c(2,2,2,2,2,2)
a3 <- a+a2
a3
```

```
## [1] 3 3 3 3 3 3
```

```
#Suma de un vector y un escalar
b <- c(1,2)
b2 <- 10
b3 <- b+b2
b3
```

```
## [1] 11 12
```

Pero veamos lo que pasa si se suman dos vectores de longitudes diferentes.

```
a
```

```
## [1] 1 1 1 1 1 1
```

```
b
```

```
## [1] 1 2
```

```
c <- a+b
c
```

```
## [1] 2 3 2 3 2 3
```

En términos matemáticos esto es incorrecto, sin embargo, en este caso, **R** expande el vector más corto hasta igual el vector más largo. En este caso, el vector *a* tiene 6 elementos, mientras que el vector *b* tiene 2 elementos. Esto es lo que llamamos reciclaje. Esto lo realiza **R** de manera silenciosa, a menos que la longitud del vector más largo no sea un múltiplo entero de la longitud del vector más corto. En estos casos deberán observar una advertencia.

Las funciones vectorizadas en *tidyverse* mostrarán errores cuando se recicla un vector que no sea un escalar.

Si se desea reutilizar, se debe hacer reciclaje de manera explícita con la función *rep()*:


```
# tibble(x=1:10, y=1:2)
# Error: Tibble columns must have compatible sizes. * Size 10: Existing data.
#* Size 2: Column `y`. i Only values of size one are recycled.
tibble(x = 1:10, y = rep(1:2,5))
```

```
## # A tibble: 10 x 2
##          x     y
##          <int> <int>
## 1        1     1
## 2        2     2
## 3        3     1
## 4        4     2
## 5        5     1
## 6        6     2
## 7        7     1
## 8        8     2
## 9        9     1
## 10       10    2
```

```
tibble(x = 1:10, y = rep(1:2, each=5))
```

```
## # A tibble: 10 x 2
##          x     y
##          <int> <int>
## 1        1     1
## 2        2     1
## 3        3     1
## 4        4     1
## 5        5     1
## 6        6     2
## 7        7     2
## 8        8     2
## 9        9     2
## 10       10    2
```

Observa que en el segundo caso evitamo el error pidiendo que la secuencia de 1 al 2 se repita 5 veces, con esto se iguala la longitud entre x, y y es posible constuir el data frame deseado.

3.7 Nombrar vectores

Todos los vectores pueden ser nombrados. Se puede asignar un nombre al momento de crearlos:

```
x ← c(a=1, b=2, c=3)
x
```

```
## a b c
## 1 2 3
```

También, un vector ya creado, puede nombrarse utilizando la instrucción set_names():

```
set_names(x, c("col w", "col x", "col y"))
```

```
##    col w     col x     col y
##    1         2         3
```

3.8 Subconjuntos

En capítulos anteriores aprendimos que en caso de ser necesario filtrar filas o columnas de un *data frame* utilizamos la función *filter()*. Pero esta instrucción solo funciona con objetos de tipo *data fame*. Si trabajamos con vectores, necesitamos utilizar los corchetes *[.]* para especificar los elementos que deseamos conservar:

Por ejemplo;

```
vector_completo ← c(10,20,30,40,50,60,70,80,90,100)
# subconjunto de los primeros 3 valores
vector_completo[1:3]
```

```
## [1] 10 20 30
```

```
# subconjunto de los últimos 3 valores
vector_completo[c(8,9,10)]
```

```
## [1] 80 90 100
```

```
# subconjunto de los últimos 3 valores, forma general
vector_completo[(length(vector_completo)-2):length(vector_completo)]
```

```
## [1] 80 90 100
```

```
# subconjunto excluyendo el primer elemento
vector_completo[c(-1)]
```


4 Vectores recursivos / Listas

Las listas son un poco más complicados que los vectores atómicos. Como vimos en los ejemplos anteriores una lista puede contener otras listas. Sin embargo, las listas son adecuadas para representar estructuras jerárquicas o de tipo árbol.

Para crear una lista utilizamos la función *list()*:

```
estudiantes ← list(c("Maria","Juan","Pedro", "Carmen", "Magnolia", "Artu-
ro"),
           c(20,25,28,28,25,29)) %>%
     set_names( "Nombre", "Edad")
estudiantes
```

```
## $Nombre
## [1] "Maria" "Juan" "Pedro" "Carmen" "Magnolia" "Arturo"
##
## $Edad
## [1] 20 25 28 28 25 29
```

Observa que hemos creado una lista con dos elementos y asignamo nombre a cada uno de estos elementos usando **set_names()**.

Podemos utilizar la función str() para identificar el tipo de elementos y la estructura de cada lista:

```
str(estudiantes)
```

```
## List of 2
## $ Nombre: chr [1:6] "Maria" "Juan" "Pedro" "Carmen" ...
## $ Edad : num [1:6] 20 25 28 28 25 29
```

Esto indica que la lista tiene dos elementos, uno bajo la etiqueta nombre y otro *edad*

4.1 Subcojuntos

Para extraer elementos de una lista tenemos dos opciones, utilizar una vez los corchetes [] o utilizar dos veces los corchetes *[[]]*. Existe una distinción entre ambas formas.

Si utilizamos una vez los corchetes *[]* obtendremos por resultado una lista. Mientras que si utilizamos dos veces los corchetes, obtendremos por resultado un elemento de una lista. Veamos un ejemplo.

La lista estudiantes esta conformada por dos listas, la primera que contiene el nombre de los estudiantes y la segunda sus edades.

```
View(estudiantes) # Recuerda que esta función abre el objeto y lo muestra en
la ventana
```

```
## Warning in system2("/usr/bin/otool", c("-L", shQuote(DSO)), stdout = TRUE):
## running command ''/usr/bin/otool' -L '/Library/Frameworks/R.framework/Re-
sources/
## modules/R_de.so'' had status 1
```

```
# Primer elemento de la lista estudiantes es el cual es a su vez una lista
estudiantes[1]
```

```
## $Nombre
## [1] "Maria" "Juan" "Pedro" "Carmen" "Magnolia" "Arturo"
```

```
# El segundo elemento de la lista estudiantes es el cual es a su vez una lis-
ta
estudiantes[2]
```

```
## $Edad
## [1] 20 25 28 28 25 29
```

Si queremos obtener el primer elemento de la lista de nombres debemos utilizar doble corchete para especificar la primer lista y luego corchete simple para el elemento de esa lista, especifica:

```
estudiantes[[1]][1]
```

```
## [1] "Maria"
```

```
#También podemos usar el nombre de la lista que se asignó anteriormente
estudiantes[["Nombre"]][1]
```

```
## [1] "Maria"
```

```
estudiantes
```

```
## $Nombre
## [1] "Maria" "Juan" "Pedro" "Carmen" "Magnolia" "Arturo"
##
## $Edad
## [1] 20 25 28 28 25 29
```


Para obtener el nombre y las edades de los primeros dos estudiantes:

```
estudiantes[[1]][1:2]
```

```
## [1] "Maria" "Juan"
```

```
estudiantes[[2]][1:2]
```

```
## [1] 20 25
```

5 Atributos

Cualquier vector puede contener información adicional a través de sus atributos. Se puede considerar a los atributos como una lista de vectores con nombre que puede ser adjuntas a cualquier otro objeto.

Para obtener y definir atributos se utiliza la función attr() o para ver todos a la vez, utilizamos *attributes()*:

```
x ← 1:10
typeof(x)
```

```
## [1] "integer"
```

```
attr(x, "Saludo")←"Hola"
attr(x, "Despedida")←"Adiós"
attributes(x)
```

```
## $Saludo
## [1] "Hola"
##
## $Despedida
## [1] "Adiós"
```

En este ejemplo hemos construido dos atributos para el vector x, el primer atributo es “Saludo” y tiene contenido el valor de “Hola”. El segundo tien un atributo “Despedida” y tiene el valor de “Adiós”.

Existen tres atributos importantes comúnmente utilizados, estos son: *Nombres*; utilizados para nombrar los elementos de un vector. *Dimensiones*; hacen que un vector se comporte como una matriz o un arreglo y *clase*; utilizada para implementar el sistema orientado en objetos de **R**.

6 Vectores aumentados

Los vectores aumentados son vectores atómicos y/o listas que contienen vectores con atributos adicionales, incluyendo una clase. Los vectores aumentados se comportan diferentes a los atómicos que los constituyen. Los más importantes son: factores, fechas y tibbles

6.1 Factores

Ya hemos trabajado con factores anteriormente, los cuales usamos para trabajar con variables categóricas. Los factores están construidos con números enteros:

```
x <- factor(c("a","b","c"), levels = c("a","b","c"))
typeof(x)
```

```
## [1] "integer"
```

```
attributes(x)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

6.2 Fechas

Las fechas en **R** son vectores numéricos que representan fechas.

```
x <- as.Date("2020-01-01")
unclass(x)
```

```
## [1] 18262
```

```
typeof(x)
```

```
## [1] "double"
```

```
attributes(x)
```

```
## $class
## [1] "Date"
```

En capítulos anteriores ya hemos produndizamos en el uso de fechas.

6.3 Tibbles

Tibbles son listas aumentadas, pueden ser de tres clases: * *tbl_df* * *tbl* * *data.frame*

Tienen dos atributos: nombres de columnas, el tipo y los nombre de filas

```
data <- data.frame(x=1:5, y=6:10)
typeof(data)
```

```
## [1] "list"
```

```
attributes(data)
```

```
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4 5
```

En capítulos anteriores ya hemos profundizado en otros atributos de los objetivos *tibble* y *data frame*

7 Actividades

1. Construye un vector que contenga tres elementos; el primero que sea una lista con números que vayan de dos en dos hasta el 200. El segundo elemento que sea las letras a las que se puede acceder en **R** con **letters.** Finalmente el tercer elemento que sea el número 8. Propón un nombre cada elemento.

2. Construye una función que pueda convertir el contenido de un vector que es del tipo double en tipo *entero*. Apóyate en la función *round()*

3. Crea una función para cada uno de los casos

• Consuma un vector y regrese el último valor

• Consuma un vector y regrese el elemento en la posición 7

• Consuma un vector y regrese solo los números que sean pares

4. Construye la siguiente lista recursiva. Una vez creada modifica lo siguiente;

• Cambia “a” por “z”

• Cambia “f” por “w”

• Agrega un cuarto elemento que sea un vector con los números del 5:10

5. Construye una lista que tenga tres elementos. El primero de ellos que sea un data frame con tres columnas y 100 filas. El segundo elemento que sea un vector con una secuencia del 1 al 10. Finalmente el tercer elemento que sea una lista con los elementos “w”, “y”. Da un nombre adecuado a cada elemento e indica como sería posible acceder al contenido total de cada uno de los elementos.

Capítulo 13

Iteración

Capítulo 13 | Iteración

1 Introducción

Hemos aprendido que una función nos permite simplificar una tarea que deseamos realizar de manera repetida. En este capítulo nos enfocaremos en una herramienta que nos proporcionará nuevos elementos para realizar una tarea de manera repetida. Comenzaremos con el uso de bucles, definiendo los elementos básicos que deben contener, desarrollaremos algunos ejemplos un tanto complejos y aprenderemos herramientas de la librería *purrr* que nos ayudarán a simplificar cada vez más determinados procesos. La repetición de tareas, utiliza dos tipos programación **imperativa** y funcional' en lo que sigue describimos cada una de ellas.

1.1 Previos

Usaremos las librerías de costumbre e incorporaremos una nueva. *Purrr,* la que nos proporciona herramientas que permiten simplificar el trabajo con ciclos y repetición de tareas.

```
library(tibble)
library(purrr)
library(tidyverse)
```

2 Programación Imperativa

Dentro de este tipo de programación identificamos los bucles comunes, los cuales si ya tienes algo de experiencia con otros software de programación, conocerás como **for** y **while**.

El uso de un bucle **for**, nos permite ejecutar una acción que se repite mas de una vez. Es una opción alternativa, y en ocasiones complementaria a lo que hacemos cuando creamos funciones. La

idea detrás de un bucle **for** es la iteración que será de mucha utilidad cuando necesitamos realizar una tarea con múltiples entradas en diferentes columnas o diferentes conjuntos de datos.

2.1 Bucles con for

Un bucle con **for** indica que deseamos que una misma acción se repita desde un valor de inicial hasta un valor final y tiene tres componentes;

1. **Salida** Donde se almacenará el resultado del bucle
2. **Secuencia** Es la indicación sobre qué iterar. En cada iteración se asigna a una variable de referencia, digamos **i**, un valor de inicio y de fin. Este valor esta relacionado con el tamaño de una secuencia.
3. El *contenido* se refiere a la función del código desarrollado. Es decir ¿para qué construimos el bucle?

Veamos estos elementos con un ejemplo. Generemos primer una tabla con datos que contenga valores aleatorios y llamemos a esta tabla **dp**.

```
dp ← tibble(
a = rnorm(100),
b = rnorm(100),
c = rnorm(100),
)
```

Imagina que queremos la suma (no la cantidad de números) de todos los números que sean menores que cero contenidos en **dp**. Esto requiere que vayamos a cada uno de los números presentes en la tabla de datos, que los cataloguemos como positivos o negativos y que sumemos todos aquellos que sean negativos.

Para resolver este problema usaremos varias de las técnicas que hemos aprendido con anterioridad.

Construyamos un vector alternativo en el que almacenemos sólo los números negativos y se asigne cero al resto, de esta manera podremos simplificar el problema, únicamente a resolver una suma. La dificultad de este proceso radicará en la primera parte.

Empecemos haciendo una función que analice un número, si es negativo que regrese el mismo número, mientras que si es positivo que regrese cero.

```
negativo←function(x){
     if(x<0) x
     else 0
}
negativo(-5)
```

```
## [1] -5
```

```
negativo(5)
```

```
## [1] 0
```

Ahora, construyamos el vector alternativo para la primera columna de la tabla de datos dp. Este vector será una copia de la primera columna, excepto que cambiará los valores positivos por cero, dejando los valores negativos sin cambio.

```
alter←vector("double", nrow(dp))
for (i in seq(1:nrow(dp))) {
     alter[[i]]←as.numeric(negativo(dp[[i,1]]))
}
```

Observa que el primer elemento del bucle salida se declaró usando **alter<-vector("double", nrow(-dp))** que es donde guardaremos de manera provisional los datos. Esto indica que se construya un vector con datos del tipo *double* y con una dimensión equivalente al número de filas en **dp**.

La *secuencia* del bucle esta determinada por **i in seq(1:nrow(dp))** indicando que se recorran todos los números desde el 1, hasta el valor contenido en **nrow(dp)**, es decir, 1,2,3, etc.

Finalmente, el *contenido* del bucle esta formado en este caso sólo por la instrucción:

```
alter[i]←as.numeric(negativo(dp[i,1])).
```

Donde se establece que en el vector alternativo se guarde el resultado de la función *negativo* previamente construida.

Con este proceso, vector alternativo, contiene ceros o los valores negativos. Por lo que simplemente los sumamos.

```
alter
```

```
## [1]  0.00000000  -0.10176134 -2.23352886 0.00000000 0.00000000 -0.54460437
## [7]  0.00000000   0.00000000 0.00000000 -0.24517022 0.00000000 0.00000000
## [13] -0.85376544 -0.38268287 -0.38035669 -0.48677457 0.00000000 0.00000000
## [19] 0.00000000   0.00000000 -0.62300656 -0.42153973 -0.88072216 -0.87711360
## [25] 0.00000000   0.00000000 -1.56144885 0.00000000 0.00000000 -0.20533629
## [31] 0.00000000  -0.51738903 0.00000000 0.00000000 0.00000000 0.00000000
## [37] -1.84221170  0.00000000 0.00000000 0.00000000 0.00000000 -0.71859971
```



```
## [43] 0.00000000  -0.47424284 0.00000000 0.00000000 0.00000000 -0.20075857
## [49] 0.00000000   0.00000000 -1.03756647 -0.04174784 0.00000000 0.00000000
## [55] -1.19504921 -0.39162325 -1.14405728 -0.34322336 0.00000000 0.00000000
## [61] -2.23358140  0.00000000 0.00000000 0.00000000 -0.92486065 -1.36618327
## [67] -0.52160677  0.00000000  0.00000000 -0.84424244 -0.75822839 0.00000000
## [73]  0.00000000  0.00000000  0.00000000 -1.08875574 -0.71136932 -0.08702757
## [79] -0.13780567  0.00000000 -0.46719026  0.00000000  0.00000000 -1.47158280
## [85] -0.20381224  0.00000000 -0.85813267 0.00000000  0.00000000 0.00000000
## [91]  0.00000000  0.00000000  0.00000000 0.00000000 -0.03396818 -0.39447631
## [97]  0.00000000 -1.44577392  0.00000000 -0.40321559
```

```
sum(alter)
```

```
## [1] -31.65609
```

Este es el resultado de la suma de los valores negativos de la primera columna. Deberíamos hacer lo mismo para el resto de las columnas y al final sumar todo. Esto implica copiar y pegar nuevamente el código y ajustarlo para cada columna. Otra opción es hacer uso de lo que hemos aprendido.

Construyamos otra función que regrese de manera directa la suma que obtuvimos con anterioridad.

```
sum_neg←function(x){
     alter←vector("double", nrow(x))
     for (i in seq(1:nrow(x))) {
     alter[[i]]←as.numeric(negativo(x[[i,1]]))
     }eturn(sum(alter))
     invisible(x)
}
```

Observa que la función contiene los mismos elementos que consideramos anteriormente, con la diferencia que al estar escritos como función, debemos incluir la opción **return**. Probemos nuestra función con la primera columa de los datos de **dp**.

```
sum_neg(dp[1])
```

```
## [1] -31.65609
```

Comprobamos que hemos obtenido el mismo resultado. Para el resto de las columnas, podemos simplemente construir un nuevo ciclo, el evalua la función **sum_neg()** sobre cada una de las difernetes columnas.

```
columna←vector("double", ncol(dp))
for (i in seq(1:ncol(dp))) {
     columna[[i]]←sum_neg(dp[i])
}
columna
```

```
## [1] -31.65609 -39.87182 -33.76040
```

Observa que ahora el vector de salida del ciclo tiene una dimensión dada por **ncol(dp)** y en cada una de ellas se almacenará el contenido de la suma de la columna.

Finalmente, al sumar todos los contenidos de columna tendremos el resultado final.

```
sum(columna)
```

```
## [1] -105.2883
```

Podemos mejorar la función *sum_neg* haciendo un doble bucle. Pues hemos visto que en el proceso de solución fue necesario hacer dos bucles. Considera la función *sum_neg_mejor*

```
suma_neg_mejor←function(x){
     columna←vector("double", ncol(x))
     alter←vector("double", nrow(x))
     for (j in seq(1:ncol(x))) {
     for (i in seq(1:nrow(x)) ) {
     alter[[i]]←as.numeric(negativo(x[[i,j]]))
     }
     columna[[j]]←sum(alter)
     }
     return(sum(columna))
}
```

Si observas con detalle la función contiene todos los elementos que hemos efectuado en partes. De esta manera tenemos que;

```
suma_neg_mejor(dp)
```

```
## [1] -105.2883
```

Estos son los elementos básicos en la construcción de un bucle.

Observa que en el ejemplo anterior la secuencia sobre la cual se definió el bucle establece que un contador el cual toma valores dentro de un rango dado por una secuencia y va desde 1 hasta **nrow(x)**

el cual corresponde a un valor numérico. Para profundizar en esto consideremos lo siguiente;

```
for (i in seq(1:ncol(dp))) {
print (i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

```
class(i)
```

```
## [1] "integer"
```

Vemos que el *contador* i toma valores, comenzando en 1 y hasta 3, precisamente por son tres las columnas contenidas en **dp**. Para referirnos a una posición cuando usamos este contador y extraer o asignar un valor debemos usar doble corchete. Las razones de ello las explicamos en el capítulo anterior. Nota que en este caso *i*, es un número entero.

Una forma alternativa de hacer esto es usar la función **names()**. Observa

```
for (i in names(dp)) {
print (i)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
```

En este caso la variable i es de tipo carácter y toma valores de texto, no numéricos. Esta forma de asignación de la secuencia nos puede resultar útil cuando mas que desear llevar un contador, deseamos efectuar acciones sobre variables con determinados nombres. La opción más común es usar la secuencia sobre los índices con valores numéricos.

Anteriormente comentamos que el primer elemento para el desarrollo de un bucle es la identificación de la salida. Esto supone conocer el tamaño que tendrá. En nuestro ejemplo definimos **columna<-vector("double", ncol(x))** pues sabíamos que por cada columna en el *data frame* original, necesitaríamos una columna para sustituir los valores positivos por cero. En caso de que no conozcamos cual será el tamaño de la *salida*, podemos usar una función que defina únicamente el tipo de dato contenido sin incluir una especificación sobre su dimensión. Si este es el caso podemos hacer uso de listas.

Considera que estamos realizando una simulación y que es necesario generar la media de un

conjunto de datos aleatorios, y que nuestra limitante es que cada uno de esos conjuntos debe contener diferente cantidad de datos. Supón que necesitaremos 5 conjuntos de datos. En este caso debido a que cada conjunto de datos es diferentes, no podemos establecer la dimensión de la salida. Para resolver este problema hagamos lo siguiente;

- Creemos un vector de salida que tenga diferentes dimensiones dependiendo de la cantidad de datos que se generen.
- Generemos un número aleatorio para definir cuantos datos queremos que se generen
- Generemos los números aleatorios

```
n_conj ← 5 # Aquí tenemos el número de conjuntos que necesitamos
conjuntos←vector("list", n_conj) #Aquí creamos la lista que contendrá los
cinco conjuntos de datos.
# Observa que no sabemos cuantos números habrá en cada conjunto de datos.
#Sólo sabemos que habrá 5 conjuntos
for (i in 1:n_conj) {
      n ← sample(1000,1)
      #Aquí estamos generando de manera aleatoria un número que definirá
      #la cantidad de datos de cada conjunto
      conjuntos[[i]] ← rnorm(n, mean=5, sd=1)
      # Aquí estamos diciendo que se cree el conjunto i, con i=1,2,3,4,5.
      #Cada conjunto debe contener n números aleatorios (n varia) de manera
      #que tengan media 5 y desviación estándar 1
}
str(conjuntos)
```

```
## List of 5
## $ : num [1:959] 3.94 4.3 7.73  4.47 4.9 ...
## $ : num [1:21]  5.54 6.24 5.42 5.42 3.44 ...
## $ : num [1:813] 3.19 4.3  6.08 4.29 5.06 ...
## $ : num [1:362] 6.3  5.21 4.71 4.38 4.34 ...
## $ : num [1:919] 5.12 6.03 5.79 5.89 6.1 ...
```

Observa que cada uno de los conjuntos tiene un número diferente de datos. Todos estos datos, comparten dos características, su media debe ser aproximadamente cinco y su desviación estándar uno, pues así definimos su creación. Para comprobar esto, podrías pensar que es suficiente hacer **mean(conjuntos)**. Sin embargo, esto generará un error pues el objeto conjuntos es una lista y debe ser tratada como tal. Para ello debemos indicar;

```
for (i in 1:n_conj) {
cat(mean(conjuntos[[i]]), sd(conjuntos[[i]]), "\n")
}
```



```
## 4.970587 0.9733361
## 5.302685 0.8131152
## 5.013924 0.987392
## 5.029571 0.9541631
## 4.950029 1.013569
```

Este ejercicio además nos ayuda a entender como funciona el comando **rnorm()** el cual sugiere que aun cuando tengamos diferentes cantidades de datos, se esforzará porque esos datos cumplan la característica indicada en torno a la media y a la desviación estándar.

Una forma alternativa de solucionar este problema es;

```
for (i in 1:n_conj) {
conjunto ← double()
n ← sample(1000,1)
conjunto←c(conjunto, rnorm(n, mean=5, sd=1))
cat(length(conjunto), mean(conjunto), sd(conjunto), "\n")
}
```

```
## 987 4.994121 0.9667868
## 956 4.962973 0.9772604
## 411 4.997763 1.044606
## 521 4.98704  1.012902
## 720 5.001593 1.002405
```

Observa que ahora hemos creado una variable de nombre conjunto con una dimensión desconocida y solo hemos dicho que se trata del tipo **double()**. De esta manera el tamaño se adapta a contender la cantidad de valores aleatorios que se generen. Hemos pedido que se muestre en pantalla la longitud del conjunto para que puedas corrobarar que en cada caso es diferente.

En ejemplo anterior hemos supuesto que desconocemos, la longitud del vector donde se guardarán los datos de salida del bucle. Ahora, supongamos que no conocemos la longitud de la secuencia. Es decir, en los ejemplos anteriores hemos pedido que el bucle de efectúe por ejemplo **1:n_conj** para decir que comience en uno y termine en el valor de **n_conj** de la misma manera hemos establecido secuencias para ello. ¿Qué sucede si no conocemos con exactitud hasta donde debe llegar la secuencia, pero si sabemos que el bucle se tiene que repetir indefinidamente *mientras* que se cumpla determinada condición? En estos casos hacemos uso de la función *while()*. Un bucle con *while()* simplemente ordena que se efectúe una actividad siempre que se cumpla determinada condición.

Por ejemplo, imagina que tienes un dado, y que deseas saber cuantas veces es necesario lanzarlo para obtener dos veces seguidas la cara con seis puntos.

Construyamos primero la función que llamaremos dado, la cual cada vez que se ejecute regresará un valor del 1 al 6, haciendo referencia a las caras de un dado.

```
dado ← function(){
      sample(c("1","2", "3", "4", "5","6"),1)
}
dado()
```

```
## [1] "2"
```

Queremos que esta función se ejecute hasta que obtengamos dos veces seguidas seis puntos.

```
n_seis ← 0
veces ← 0
while (n_seis<2) {
      a← dado()
      if (a=="6"){
            n_seis ← n_seis+1
      } else {
            n_seis ← 0
      }
      veces ← veces+1
}
veces
```

```
## [1] 15
```

Observa que hemos inicializado dos variables en cero, la primera **n_seis** que contabiliza el número de ocasiones que se obtienen seis puntos y **veces** que contabiliza el número de veces que se ha lanzado el dado. La condición **n_seis<2** indica que el bucle se repita hasta que se cumpla esta condición. En este caso el bucle esta acompañado de una condicional. La finalidad de la condicional es cuidar que la condición **dado()=="6"** se cumpla dos veces seguidas. Siempre que se obtenga un seis, pero en el siguiente lanzamiento, no se obtenga, la variable **n_seis** se reinicia a cero. El bucle se repetirá hasta el cumplimiento de la condición. Si deseas ver los posibles resultados de cada lanzamiento para verificar nuestro resultado, puedes agregar **print(a)** justo después de definir **a=dado().**

3 Programación Funcional

Gracias a que **R** es un lenguaje de programación funcional, los bucles no son algo tan usado y necesario y muchas veces su uso se puede sustituir por funciones. Por ejemplo, considera que tienes una tabla de datos y que te interesa obtener, la media, la mediana y la varianza de cada una de las filas de la tabla de datos. Una opción sería crear un bucle que calcule la media de cada fila, luego repetirlo para que calcule la mediana y finalmente en la varianza.

Usemos estos datos para el ejemplo.

```
dp2 ← tibble(
      a = rnorm(5),
      b = rnorm(5),
      c = rnorm(5),
      d = rnorm(5),
      e=a*b,
      f=c*d,
      g=c*f
)
```

Para el caso de la media tendríamos

```
medias ← vector("double",nrow(dp2))
for (i in 1:nrow(dp2)) {
      medias[[i]] ← mean(as.numeric(dp2[i,]))
}
medias
```

```
## [1] -0.33178725 -0.86442835 -0.43945491 0.01765847 -0.24148979
```

Para obtener las otras estadísticas deberíamos copiar y pegar el código, cambiando el estadístico a calcular por **var()** para la varianza y **median()** para la media. Esto implica copiar y pegar el código, lo cual como ya sabemos aumenta las posibilidades de cometer errores. En lugar de ello podemos construir una función

```
opera_fila ← function(x, funcion){
res ← vector("double",nrow(x))
for (i in 1:nrow(x)) {
      res[[i]] ← funcion(as.numeric(x[i,]))
}
      return(res)
}
opera_fila(dp2, mean)
```

```
## [1] -0.33178725 -0.86442835 -0.43945491 0.01765847 -0.24148979
```

```
opera_fila(dp2, var)
```

```
## [1] 0.6806436 1.6527852 0.8852387 0.4776814 0.2369013
```

De esta forma podemos simplemente cambiar el tipo de función y operar sobre la función que deseemos. Observa que estamos operando una función dentro de otra función. Estas es una de las opciones que caracterizan a **R** como un programa de operación funcional, gracias al cual, podemos disminuir el uso de bucles. En específico la librería *purrr* contiene funciones que de una manera sencilla permiten operar los bucles *for* mas comunes.

4 Librería Purrr

Purrr es una librería que permite simplificar el manejo de los ciclos *for* de uso mas común. La base de esta librería consiste en el trabajo con funciones de tipo *map.*

4.1 Mapeos un argumento

Una función de este tipo aplica una función de forma iterativa a cada elemento de una lista o vector. Las funciones de tipo *map* son;

- *map()*
- *map_lgl()*
- *map_int()*
- *map_dbl()*
- *map_chr()*

Consideremos el caso del ejemplo anterior donde calculamos las estadísticas de un conjunto de datos por fila. Supón ahora que deseamos obtener la información relativa a la media, mediana y varianza pero ahora por columna. Para ello hacemos;

```
map_dbl(dp2,mean)
```

```
##           a           b           c           d           e           f
##    -0.91892372 -0.07381384 -0.66879610 -0.38877373 -0.12695758 0.06646295
##           g
##    -0.49250055
```

Observa que en este caso la función **map_dbl** esta calculando la media de cada uno de las vectores (columnas) que hay dentro del objeto **dp2**. Hemos usado **map_dbl** pues cada elemento dentro de columna, constituye un objeto de tipo double y por tanto nuestro resultado será del mismo tipo.

Con esta función podemos calcular el resto de las estadísticas;

```
dp2 %>% map_dbl(median)
```

```
##           a           b           c           d           e           f
##    -1.05556391 -0.13206132 -0.30770291 -0.66936001 0.05454626 -0.26098892
##           g
##    -0.24800949
```

```
dp2 %>% map_dbl(var)
```

```
##          a         b         c      d         e         f         g
##    1.2236594 0.4771132 1.9820218 0.7810474 0.4936036 0.4015788 0.3424782
```

Podemos aplicar las tres funciones al mismo conjunto de datos haciendo uso de la función **invoke_map**

```
funciones<- list(mean,median,var)
invoke_map(funciones,x=as.numeric(dp2[[1]]))
```

```
## [[1]]
## [1] -0.9189237
##
## [[2]]
## [1] -1.055564
##
## [[3]]
## [1] 1.223659
```

Observa que estos corresponde con los resultados de la primera columna. Entonces **invoke_map** permite aplicar diferentes funciones a un vector o lista. Observa que en este caso se aplicó la función a un vector de datos y para acceder a el usamos **dp2[[1]]** pudimos haber obtenido el mismo resultado haciendo **dp2$a**, pues el objeto **dp2** es una tabla de datos.

La función *map* también se puede aplicar a elementos que son listas. Por ejemplo;

```
x <- list(a=1:10, b=5:40, c=6:23)
y <- list(a=5:10, b=20:40, c=7:12)
```

Tanto *x* como *y* contienen tres conjuntos de datos, con diferentes dimensiones. Podemos obtener la media de cada conjunto de datos de *x* y la varianza de cada conjunto de datos de *y* haciendo;

```
map(x, mean)
```

```
## $a
```

```
## [1] 5.5
##
## $b
## [1] 22.5
##
## $c
## [1] 14.5
```

```
map(y, var)
```

```
## $a
## [1] 3.5
##
## $b
## [1] 38.5
##
## $c
## [1] 3.5
```

Si quisiéramos aplicar todas las funciones al primer elemento de la lista *x*, tendríamos;

```
invoke_map(funciones,x=x[[1]])
```

```
## [[1]]
## [1] 5.5
##
## [[2]]
## [1] 5.5
##
## [[3]]
## [1] 9.166667
```

invoke_map() permite aplicar diferentes funciones a una misma lista o vector.

Las funciones del tipo **map_*** permiten indicar el tipo de objeto contenido, ya sea un vector atómico o una

lista. Por ejemplo;


```
z ← list(x = 1:3, y = 4:5)
map_int(z, length)
```

```
## x y
## 3 2
```

```
map_dbl(z, length)
```

```
## x y
## 3 2
```

4.2 Atajos

Gracias a *purrr* es posible aplicar ciertos atajos que simplifican la escritura y el trabajo con ciclos. Para ver esto, usemos una de las bases que ya hemos trabajado con anterioridad. Usaremos una muestra de la base de datos que usamos en el capítulo 3, sobre la Encuesta Nacional de Ocupación y Empleo.

Carguemos la base de datos

```
library(readxl)
setwd("~/Dropbox/Curso de R/Cap13_Iteraccion")
datos ← read_excel("mu_enoe.xlsx")
```

En el capítulo tres, de manera gráfica, establecimos una relación entre la educación y el ingreso. Imagina ahora que deseamos analizar esa relación, pero ahora usando un modelo de regresión lineal. Considera ademas que deseamos que ese modelo se efectúe para cada uno de los diferentes niveles educativos. Haciendo un conteo rápido vemos que la base contiene información sobre cuatro diferentes niveles educativos.

```
datos %>%
     group_by(niv_edu) %>%
     count()
```

```
## # A tibble: 4 x 2
## # Groups:     niv_edu [4]
## niv_edu                          n
## <chr>                            <int>
## 1 Medio superior y superior      3335
## 2 Primaria incompleta            971
## 3 Prrimaria completa             1823
## 4 Secundaria completa            4151
```

Para construir un modelo de regresión lineal para cada uno de los niveles educativos, deberíamos separar la muestra en cuatro partes y para cada una de ellas ejecutar el modelo.

Una opción mas amigable es la siguiente;

```
modelos <- datos %>%
     split(.$niv_edu) %>%
     map(~lm(ingreso_mensual~anios_esc, data=.))

length(modelos)
```

```
## [1] 4
```

El código anterior ha permitido que creemos un modelo para cada uno de los niveles educativos. Observa que la información resultante esta contenida en el objeto *modelos* el cual se trata de una lista. Como vimos, una lista es un vector que contiene información de elementos que son heterogéneos. La estimación del modelo de regresión lineal se efectúa con el comando **lm** y hemos usado una función map que efectuará el modelo sobre cada uno de los componentes en los que se ha divido la muestra total con *split*.

Observa de manera especial que hemos usado el . (punto) en este caso el punto toma el lugar del nombre de la base. El uso de este instrumento permite ciertas ventajas en la programación. La principal es la disminución en la escritura.

En un modelo de regresión lineal hay dos valores que nos interesan de manera especial. El primero de ellos es el valor estimado del coeficiente que relaciona, en este caso la educación con el ingreso. El segundo se conoce como *r cuadrada* y se interpreta como la confianza en el modelo generado. Para acceder a estos valores hacemos;

```
modelos %>%
     map(summary) %>%
     map_dbl(~.$r.squared)
```

```
##   Medio superior y superior   Primaria incompleta   Prrimaria completa
##   0.114641724                 0.020646450           0.001378529
##   Secundaria completa
##   0.016356175
```

```
modelos %>%
     map(summary) %>%
     map_dbl(~.$coefficients[2,1])
```

```
##   Medio superior y superior   Primaria incompleta   Prrimaria completa
##   1169.0711                   241.6287              -168.8630
```



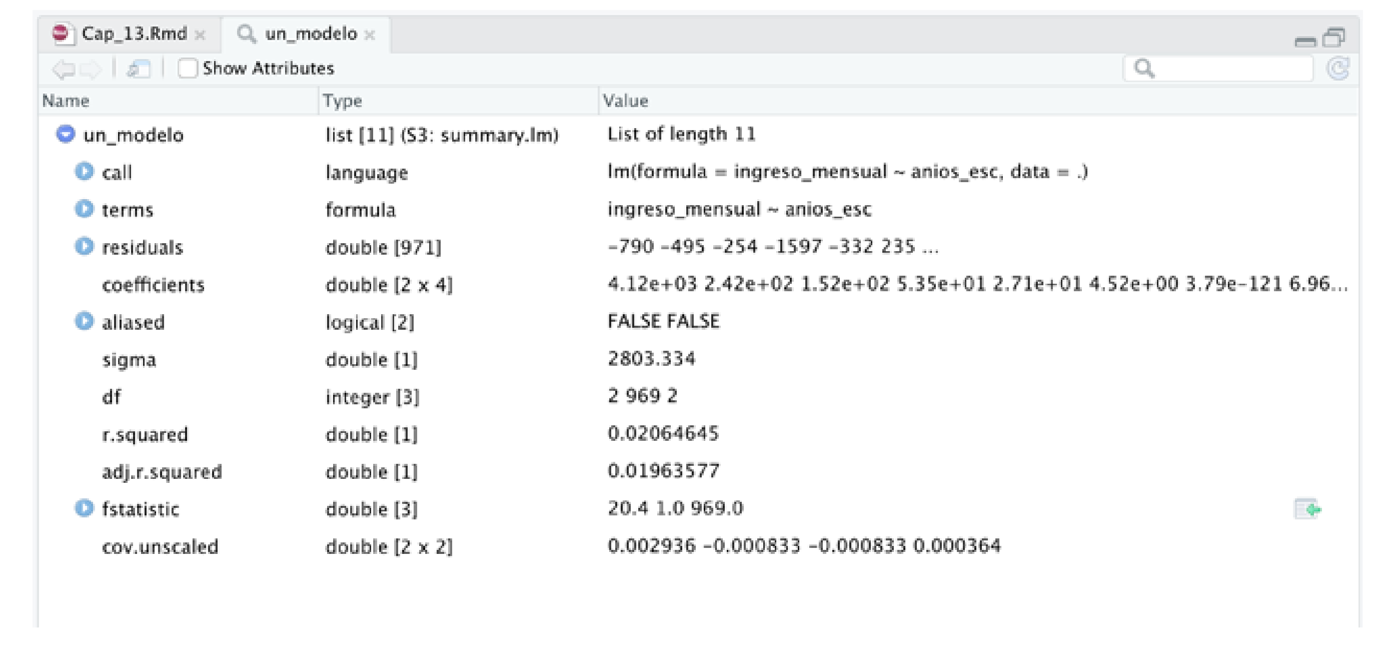
```
##   Secundaria completa
##   602.0639
```

Para entender como **R** puede acceder a la lista y regresar el valor solicitado, generemos un objeto de nombre **un_modelo** en el guardaremos el resumen de uno de los modelos

```
un_modelo <- summary(modelos$`Primaria incompleta`)
```

```
View(un_modelo)
```

Al ejecutar un **View(un_modelo)** se abre una ventana en la parte superior, la cual muestra el contenido de la lista. La cual se ve mas o menos así



Name	Type	Value
un_modelo	list [11] (S3: summary.lm)	List of length 11
call	language	lm(formula = ingreso_mensual ~ anios_esc, data = .)
terms	formula	ingreso_mensual ~ anios_esc
residuals	double [971]	-790 -495 -254 -1597 -332 235 ...
coefficients	double [2 x 4]	4.12e+03 2.42e+02 1.52e+02 5.35e+01 2.71e+01 4.52e+00 3.79e-121 6.96...
aliased	logical [2]	FALSE FALSE
sigma	double [1]	2803.334
df	integer [3]	2 969 2
r.squared	double [1]	0.02064645
adj.r.squared	double [1]	0.01963577
fstatistic	double [3]	20.4 1.0 969.0
cov.unscaled	double [2 x 2]	0.002936 -0.000833 -0.000833 0.000364

Observa que dentro de la lista hay un elemento de nombre *r.squared* del tipo *double* con dimensión 1 *[1]*. En consecuencia, para acceder a este elemento usamos el punto . con el cual se manera automática se referencia a la lista y con el uso de **$** especificamos el nombre del elemento que deseamos observar.

En el caso del elemento de nombre *coefficientes* se trata de un objeto **[2x4]** por lo que debemos especificar algo más que el nombre para acceder a él. Una inspección a este elemento nos muestra que la información contenida en el, tiene dos filas y cuatro columnas (no se incluyen los nombres). El valor de interés se ubica en la fila 2, columna 1.

```
un_modelo[["coefficients"]]
```

```
##              Estimate Std. Error    t value     Pr(>|t|)
```

```
## (Intercept)   4123.5153   151.89882   27.146460   3.792944e-121
## anios_esc      241.6287    53.46058    4.519754   6.955567e-06
```

Por ello es que en el caso del elemento "coefficients" debemos especificar la posición del valor que buscamos. Gracias a que usamos **map** y **map_dbl** pudimos hacer el mismo proceso para todos los modelos contenidos en la lista modelos de una sola vez.

Como dato, el valor obtenido significa que una persona qué en una persona con nivel educativo de primaria, tiene una retribución aproximada de 241.62 pesos por cada año de escolaridad.

En el capítulo 14, profundizaremos sobre los modelos.

Una manera alternativa para acceder a los elementos contenidos en la lista *modelos,* es referirnos a ellos directamente por su nombre, aunque esto sólo funcionará si se trata de un elemento escalar.

```
modelos %>%
     map(summary) %>%
     map_dbl("r.squared")
```

```
## Medio superior y superior      Primaria incompleta    Primaria completa
##          0.114641724            0.020646450            0.001378529
##    Secundaria completa
##          0.016356175
```

4.3 Errores comunes

Considera que tienes la siguiente lista

```
num ← list(50,30,70,90,"w")
```

Y que ejecutamos lo siguiente;

```
raiz ← map(num, sqrt)
```

```
## Error in .Primitive("sqrt")(x): non-numeric argument to mathematical func-
tion
```

Esta indicación nos mostrará un error y no se ejecutará.

Obviamente ya te disté cuenta que el problema es que dentro de la lista existe un valor w no numérico con el cual no es posible calcular la raíz cuadrada. Desearíamos qué, a pesar de este error, **R** regresará el resultado sobre aquellos valores sobre los cuales si es posible ejecutar la función.

Afortunadamente, dentro de la librería *purrr* existe el comando **safely()**. Veamos cual es su funcionamiento;

```
raiz ← map(num, safely(sqrt))
str(raiz)
```

```
## List of 5
## $ :List of 2
## ..$ result: num 7.07
## ..$ error : NULL
## $ :List of 2
## ..$ result: num 5.48
## ..$ error : NULL
## $ :List of 2
## ..$ result: num 8.37
## ..$ error : NULL
## $ :List of 2
## ..$ result: num 9.49
## ..$ error : NULL
## $ :List of 2
## ..$ result: NULL
## ..$ error :List of 2
## .. ..$ message: chr "non-numeric argument to mathematical function"
## .. ..$ call : language .Primitive("sqrt")(x)
## .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Vemos que con la ejecución de la opción **safely()** el resultado es una nueva lista que contiene 5 elementos y dentro de cada elemento hay una lista de otros dos elementos. EL primer elemento de esta segunda lista tiene por nombre result mientras que el segundo *error*. Observa que a excepción del ultimo caso con w, se obtiene un valor de **error:NULL** y se muestra un resultado correcto.

Una forma un poco mas clara de ver este resultado es transponerlo

```
raiz ← map(num, safely(sqrt))
z ← transpose(raiz)
str(z)
```

```
## List of 2
```

```
## $ result:List of 5
## ..$ : num 7.07
## ..$ : num 5.48
## ..$ : num 8.37
## ..$ : num 9.49
## ..$ : NULL
## $ error :List of 5
## ..$ : NULL
## ..$ : NULL
## ..$ : NULL
## ..$ : NULL
## ..$ :List of 2
## .. ..$ message: chr "non-numeric argument to mathematical function"
## .. ..$ call : language .Primitive("sqrt")(x)
## .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Para el caso, en el que se pide la raíz de w, se obtiene;

```
str(raiz[[5]])
## List of 2
## $ result: NULL
## $ error :List of 2
## ..$ message: chr "non-numeric argument to mathematical function"
## ..$ call : language .Primitive("sqrt")(x)
## ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

un resultado nulo, pues no es posible la función sobre w y un error que consiste en una lista de dos elementos. EL primer elemento es un mensaje indicando que se trata de un carácter, mientras que en el segundo muestra una parte del código en la que falla la función **sqrt**.

Para eliminar estos errores y poder continuar, podemos usar la lista *z*, en la que guardamos los valores transpuestos. Dentro de esta lista hay dos elementos, uno de resultados y otro de errores. De esta manera podemos enfocarnos en el trabajo únicamente con los elementos de la lista que proporcionan valores válidos.

Para ello construyamos primero un vector lógico que regrese FALSO cuando en el elemento *error* exista un valor no nulo.


```
validos ← z$error%>%
     map_lgl(is_null)
validos
## [1] TRUE TRUE TRUE TRUE FALSE
```

Ahora de la lista de resultados pedimos que sólo se muestren aquellos donde el resultado es válido (lo que es lo mismo, donde el error contenga elemento nulo)

```
z$result[validos] %>%
flatten_dbl()
## [1] 7.071068 5.477226 8.366600 9.486833
```

Mediante el uso de la indicación **flatten_dbl()** hemos pedido que se muestren únicamente los valores y no la estructura de los resultados.

Además de **safely()** *purrr* posee dos funciones que operan de manera similar y de una forma mas directa.

possibly() de manera directa ejecutará la función indicada únicamente sobre los valores en los sea posible y regresará un valor determinado (en este caso NA) sobre aquellos elementos de la lista sobre los cuales no sea posible ejecutar la función.

```
raiz ← map(num, possibly(sqrt,NA))
str(raiz)
## List of 5
## $ : num 7.07
## $ : num 5.48
## $ : num 8.37
## $ : num 9.49
## $ : logi NA
```

```
num2 ← list(5,10,-8)
raiz ← map(num2, quietly(sqrt))
z ← transpose(raiz)
str(z)
## List of 4
## $ result :List of 3
## ..$ : num 2.24
## ..$ : num 3.16
## ..$ : num NaN
```

```
## $ output :List of 3
## ..$ : chr ""
## ..$ : chr ""
## ..$ : chr ""
## $ warnings:List of 3
## ..$ : chr(0)
## ..$ : chr(0)
## ..$ : chr "NaNs produced"
## $ messages:List of 3
## ..$ : chr(0)
## ..$ : chr(0)
## ..$ : chr(0)
```

4.4 Mapeo sobre múltiples argumentos

Las funciones **map2()** y **pmap()**, permiten que se lleve una cabo una misma función sobre dos vectores o un conjunto de funciones sobre el mismo vector. De manera que con estas funciones se requieren al menos dos listas.

Considera el ejemplo desarrollado con anterioridad en el que se generaron conjuntos de valores aleatorios de diferente tamaño, pero todos con la misma características, una media de 5 y desviación estándar de 1.

```
El código que utilizamos para crearlo fue mas o menos así
n_conj ← 5
conjuntos←vector("list", n_conj)
for (i in 1:n_conj) {
     n ← sample(1000,1)
     conjuntos[[i]] ← rnorm(n, mean=5, sd=1)
}
```

Considera una variación a este problema, en la cual ahora deseamos cambiar además, la media y la desviación estándar de cada conjunto de datos.

Comencemos definiendo una lista que incluya los valores que deseamos cambiar y generando un sólo conjunto con un valor específico de números aleatorios generados, por decir n=10

```
media ← list(3,5,-3,10,0)
des_est ← list(1,4,9,4,9)
map2(media, des_est, rnorm, n=10) %>%
```



```
     str()
```

```
## List of 5
## $ : num [1:10] 2.91 2.84 5.88 3.97 4.07 ...
## $ : num [1:10] 2.87 -2.23 3.44 5.86 7.4 ...
## $ : num [1:10] -4.12 -10.19 -2.32 8.76 4.79 ...
## $ : num [1:10] 7.65 18.27 11.84 16.69 9.91 ...
## $ : num [1:10] 0.902 10.005 -5.789 -10.545 -5.044 ...
```

Si deseáramos ahora cambiar el número de datos tendríamos, primero que generar una lista con los números deseados en cada caso y usar la función **pmap()** pues recuerda que esta función permite operar mas de dos listas o vectores.

```
n_num ← list(10,20,25,50,100)
str(n_num)
```

```
## List of 5
## $ : num 10
## $ : num 20
## $ : num 25
## $ : num 50
## $ : num 100
```

Juntemos todos los parámetros que están cambiando en una sola lista y después usemos **pmap()**. En la lista de parámetros, incluyamos el nombre de cada uno de ellos, para que *R* pueda entenderlos, de no hacerlos debemos asegurarnos que los declaramos en el mismo orden según se especifica en la ayuda de la función.

```
param ← list(mean=media,n=n_num, sd=des_est)
param %>%
     pmap(rnorm) %>%
     str()
```

```
## List of 5
## $ : num [1:10] 5.13 3.06 4.03 3.76 3.18 ...
## $ : num [1:20] 2.59 4.22 2.88 9.07 2.59 ...
## $ : num [1:25] -8.11 -9.54 12.35 19.08 -5.49 ...
## $ : num [1:50] 13.45 12.59 14.6 11.33 8.53 ...
## $ : num [1:100] -4.597 3.324 -19.766 0.565 -1.614 ...
```

Observa que ahora cada conjunto de datos que hemos construido tiene una diferentes dimensión. Esta dimensión esta dada por los diferentes valores de n.

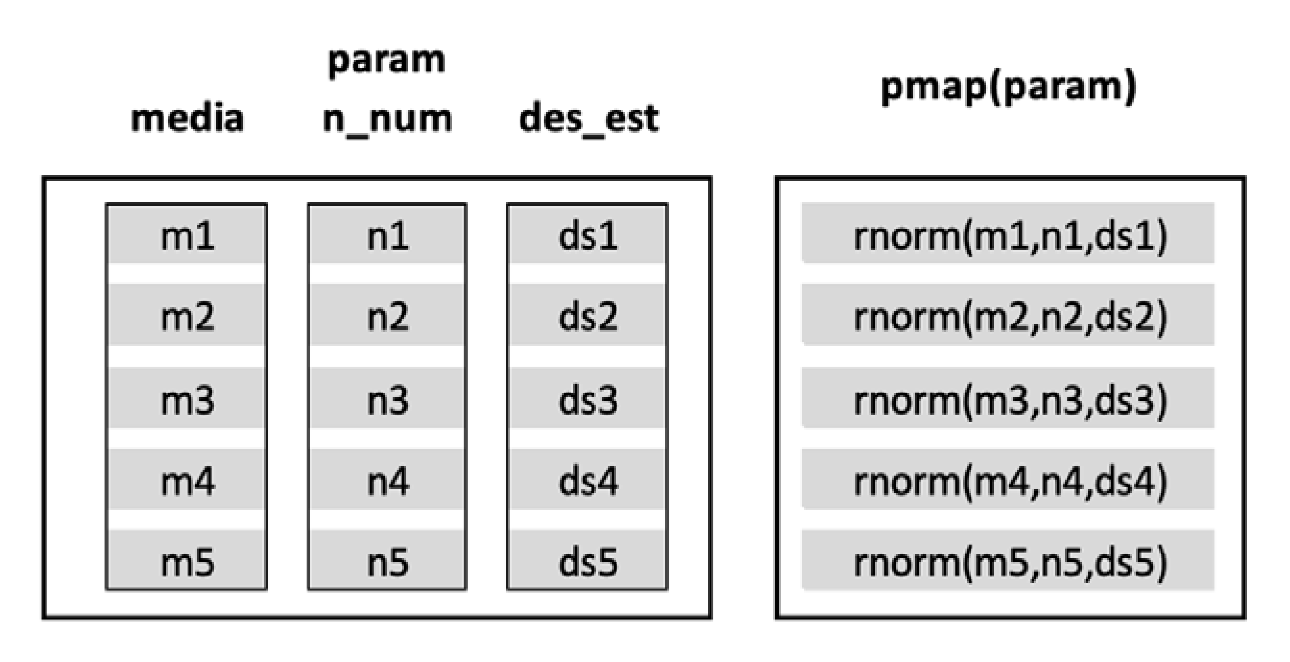
Observa que el procedimiento que realizan tanto **map2()** como **pmap()** requiere que todas las listas tengan el mismo tamaño. Es decir, tanto la media, como la desviación estándar como el número de valores a generarse deben coincidir y ser 5 opciones.

Por ejemplo, si cambiamos y solo declaramos tres opciones para n obtenemos el siguiente error

```
n_num <- list(10,20,25)
param2 <- list(mean=media,n=n_num, sd=des_est)
param2 %>%
     pmap(rnorm) %>%
     str()

# Error: Element 2 of `.l` must have length 1 or 5, not 3
```

R nos advierte que la longitud debe ser 1 o 5, pero no 2. ¿Porqué 1 sí pero 3 no? Recuerda que en el capítulo 12, vimos que **R** recicla los vectores, si sólo indicamos un valor para *n_num, R* entenderá que debe repetirlo 5 veces para que tenga la misma longitud que el resto. En cambio, si indicamos tres valores, no hay un numero entero de veces que deban repetirse esos valores para llegar a cinco. A continuación, presentamos una forma visual de entender el funcionamiento que realiza **pmap()** en el ejemplo anterior:



Anteriormente vimos que la función **invoke_map()** nos permite aplicar diferentes funciones y diferentes argumentos. Usaremos algunas de sus variaciones, por ejemplo; **invoke_map_dbl**, la cual generará resultados del tipo *double*

Por ejemplo, para aplicar una función distinta a cada elemento de un data frame, podemos hacer lo siguiente. Para el ejemplo usaremos la información contenida en datos sobre la ENOE.

```
funciones <- c("mean", "median", "sum", "median")
datos_prueba <- select(datos, edad, anios_esc, hrsocup, ingreso_mensual)
invoke_map_dbl(funciones, datos_prueba)
```

```
## [1] 56 12 444537 5375
```

Observa que hemos definido un conjunto de funciones y hemos pedido que a cada una de las cuatro columnas del data frame que contiene información de *datos_prueba* se le aplique una función diferente. De manera que el primer número resultante es la media de la edad, mientras que el segundo es la mediana de los años de escolaridad y así sucesivamente.

Podemos hacer una variación para aplicar todo un conjunto de funciones a las diferentes variables del data frame. Para ello hacemos;

```
funciones <- c("mean", "median", "IQR", "sd", "var")
datos_prueba <- select(datos, edad, anios_esc, hrsocup, ingreso_mensual)
estadisticas <- map_dfr(datos_prueba, ~invoke_map_dbl(funciones, x = .))
estadisticas$medida <- funciones
estadisticas
```

```
## # A tibble: 5 x 5
##    edad    anios_esc  hrsocup  ingreso_mensual  medida
##    <dbl>   <dbl>      <dbl>    <dbl>            <chr>
## 1  39.7    9.88       43.2     7362.            mean
## 2  39      9          45       6000             median
## 3  22      4          16       5130             IQR
## 4  14.0    4.20       16.7     5897.            sd
## 5  197.    17.6       280.     34778240.        var
```

Observa que hemos usado de manera conjunta un *map* e *invoke*. En este caso la función *map_dfr* indica que deseamos que el resultado se exprese como un *data frame.* Esta combinación permitirá que se operen todas las funciones con todas las columnas que hay en *datos_prueba.* Añadimos la opción **estadisticas$medida<- funciones** simplemente para que en el *dataframe* resultante, se puedan apreciar los nombres de cada una de las funciones que se aplicaron. Nota que el código anterior hemos incluido ~ el cual es una manera de indicar que se trata de una función, recuerda la estructura básica de **map(x, funcion)**.

La librería *purrr* tiene otras funcionalidad que nos permiten realizar una misma tarea repetidas ocasiones. Por ejemplo, considera que ahora deseamos graficar la relación que hay entre los años

de educación y el ingreso mensual, de los datos que están contenidos en la base de la ENOE y que hemos llamado *datos*. Considera que esta gráfica deseamos efectuarla para cada uno de los diferentes niveles educativos. Una opción, podría ser separa la base para cada nivel y ejecutar la instrucción de graficar cuatro veces. Una forma más amigable, es hacer lo siguiente

```
library(ggplot2)
graficas <- datos %>%
     split(.$niv_edu) %>%
     map(~ggplot(.,aes(ingreso_mensual, anios_esc))+
          geom_point())
```

Observa que hemos creado un objeto de nombre graficas que contiene el resultado. Para ello dividimos la tabla total de datos y efectuamos la gráfica. Del código anterior nota dos cosas, usamos ~ para indicar que se trata de una actividad que se repite. También usamos el punto . en lugar de indicar los datos, pues es una manera de referirnos a ellos

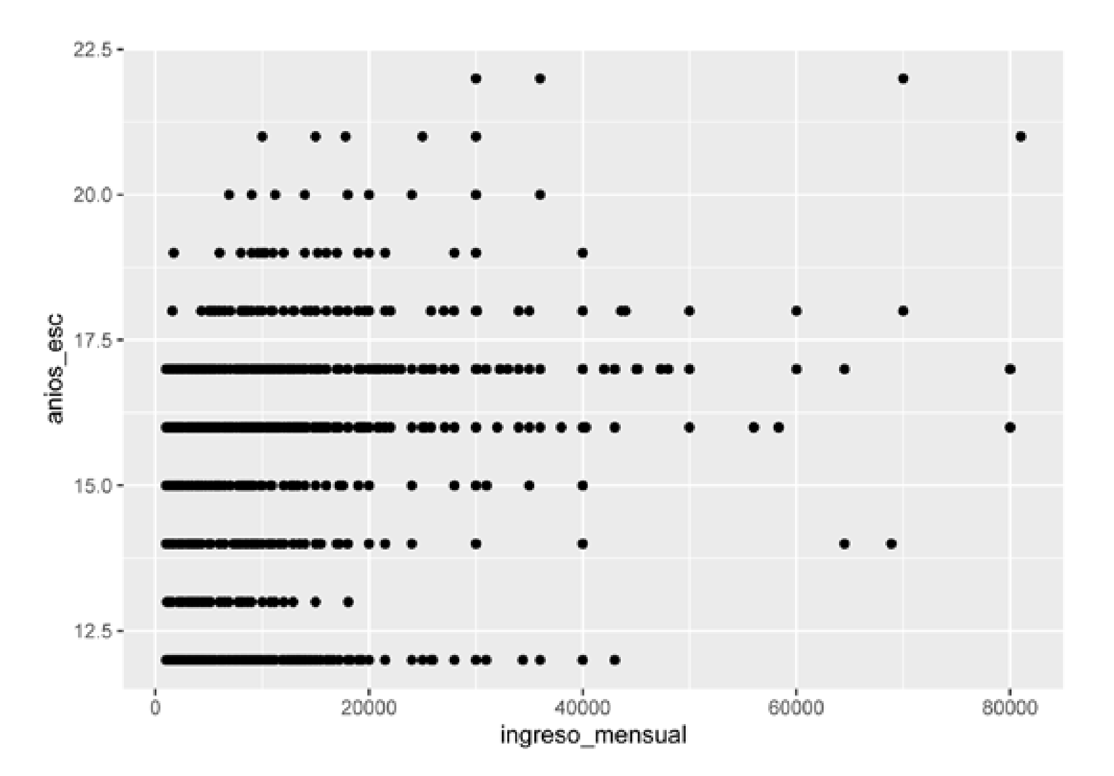
Si abrimos el objeto graficas para explorarlo nos damos cuenta que es mas o menos así

Show Attributes

Name	Type	Value
graficas	list [4]	List of length 4
Medio superior y su...	list [9] (S3: gg, ggplot)	List of length 9
Primaria incompleta	list [9] (S3: gg, ggplot)	List of length 9
Prrimaria completa	list [9] (S3: gg, ggplot)	List of length 9
Secundaria completa	list [9] (S3: gg, ggplot)	List of length 9

Observemos la primera gráfica

```
graficas$`Medio superior y superior`
```

Considera ahora que deseamos exportar estas gráficas a un archivo pdf. Para hacerlo podemos usar la función **pwalk()** la cual permite efectuar una misma operación varias veces. Antes de ver como funciona, pongamos los nombres que deseamos que tengan las gráficas

```
graf_nomb <- stringr::str_c(names(graficas), ".pdf")
graf_nomb
## [1] "Medio superior y superior.pdf" "Primaria incompleta.pdf"
## [3] "Prrimaria completa.pdf"        "Secundaria completa.pdf"
```

Para las cuatro gráficas podemos hacer lo siguiente

```
info_guardar <- list(graf_nomb, graficas)
pwalk(info_guardar, ggsave, path="~/Dropbox/Curso de R/Cap13_Iteraccion")
```

La estructura del instrucción con **pwalk()** indica que cada uno de los elementos de la lista *grafica* (que son 4), lo guardará *ggsave* en el directorio *~/Dropbox/Curso de R/Cap13_Iteraccion* con el *nombre graf_nombre*. Para que esta esta instrucción funcione de manera correcta, debe existir una correspondencia entre el tamaño de la lista *info_guardar* que son dos elementos y el tamaño de los requerimientos de la función que se ejecuta. En este caso la función *ggsave* requiere al menos dos datos, el nombre que se asignará y la gráfica a la que será asignado. Ve al directorio que estableciste y comprueba que tus gráficas se encuentran ahí.

4.5 Otros Loops

Dentro de la librería de *purrr* existen otras funciones que son útiles y facilitan una tarea repetitiva. Éstas se conocen como funciones de *predicado* o *predicate*, pues su resultado es un valor único FALSO o VERDADERO dependiendo del predicado que las acompaña.

Por ejemplo, **keep(predicado)** mantiene los elementos de una entrada donde el predicado se verdadero, mientras que **discard(predicado)** los descarta.

```
# Conservar todos los elementos donde se cumple la condición is.character==TRUE
datos %>%
     keep(is.character) %>%
     str()
```

```
## tibble [10,280 x 8] (S3: tbl_df/tbl/data.frame)
## $ estado      : chr [1:10280] "Hidalgo" "Durango" "Jalisco" "Tabasco" ...
## $ sex         : chr [1:10280] "Mujer" "Hombre" "Hombre" "Mujer" ...
## $ asiste      : chr [1:10280] "No" "No" "No" "No" ...
## $ pos_ocu     : chr [1:10280] "Trabajadores subordinados y remunerados" "Trabajadores subordinados y
## $ ing_salarios: chr [1:10280] "Más de 1 hasta 2 salarios mínimos" "Más de 3 hasta 5 salarios mínimos"
## $ niv_edu     : chr [1:10280] "Secundaria completa" "Medio superior y superior" "Medio superior y superior"
## $ num_trabajos: chr [1:10280] "Uno" "Uno" "Uno" "Uno" ...
## $ tipo_empleo : chr [1:10280] "Informal" "Formal" "Formal" "Informal" ...
```

```
# Descarta todos los elementos donde se cumple la condición is.character==FALSE
datos %>%
     discard(is.character) %>%
     str()
```

```
## tibble [10,280 x 4] (S3: tbl_df/tbl/data.frame)
## $ edad            : num [1:10280] 56 52 25 50 41 36 23 59 37 18 ...
## $ anios_esc       : num [1:10280] 12 17 15 9 17 9 16 4 9 8 ...
## $ hrsocup         : num [1:10280] 45 66 48 46 5 50 48 28 48 48 ...
## $ ingreso_mensual : num [1:10280] 5375 12900 12000 3870 1000 ...
```

Las funciones **some() every()** determinan si el predicado es verdadero para todos o solo para algunos elementos de la lista.


```
datos$sex %>%
     every(is.character) #¿Todos los datos en sexo son tipo carácter?
```

```
## [1] TRUE
```

```
combinado <- list(datos$sex, datos$edad)
combinado %>%
     every(is.character) #¿Todos los datos en la lista son tipo carácter?
```

```
## [1] FALSE
```

```
combinado %>%
     some(is.numeric) #¿Algunos datos en la lista son tipo numérico?
```

```
## [1] TRUE
```

Observa que la lista de nombre combinado está formada por los datos de columna sexo que son del tipo carácter y los datos de la columna edad que son numéricos.

Las funciones **detect** y **detect_index()** regresan el primer elemento donde el predicado es verdadero o su posición respectivamente. Considera el vector de edad que contiene una muestra de 10 valores

```
edades <- sample(combinado[[2]], 10)
edades
```

```
## [1] 18 31 17 49 62 47 35 48 23 21
```

Con **detect()** podemos encontrar el primer valor contenido en edades, que es mayor a 40, mientras que con **detect_index()** encontramos su posición.

```
edades %>%
     detect(~.>40) # ¿Cuál es
```

```
## [1] 49
```

```
edades %>%
     detect_index(~.>40)
```

```
## [1] 4
```

Por otro lado, la función **head_while** nos regresará los primeros valores mayores a 40. Si el primer valor es falso, regresará valor cero. Por otro lado, **tail_while** hace lo mismo, pero con los elementos del final de la lista. Si el último elemento no cumple la condición regresará cero.

```r
edades %>%
     head_while(~.>40)
```

```
## numeric(0)
```

```r
edades %>%
     tail_while(~.>40)
```

```
## numeric(0)
```

Finalmente, la función reduce toma una función *f* con dos argumentos y una lista o vector el cual se desea reducir usando la función *f*. Por ejemplo, si tenemos el siguiente vector numérico

```r
x <- seq(0,10)
x
```

```
## [1] 0 1 2 3 4 5 6 7 8 9 10
```

Y nos interesa determinar la suma acumulada, de manera que el último termino es la suma de todos los números.

```r
Reduce(f = "+", x , accumulate = TRUE)
```

```
## [1] 0 1 3 6 10 15 21 28 36 45 55
```

Esta función tiene muchas mas aplicaciones. Por ejemplo, podemos usarla para unir diversos *dataframe* que estén contenidos en una lista

```r
df1 <- data.frame(ID = c(1, 2, 3,4), Nombre = c("Pepe", "Juan", "Luis", "Pe-
dro"))
df2 <- data.frame(ID = c(1, 2,3,4), Inicio_Compra = c(2011, 2017, NA , 2000))
df3 <- data.frame(ID = c(1, 2, 4),
                Utlima_Compra = c("2017-03-03", "2014-03-01", "2017-05-30"),
                Pagos = c(46690, 25456,25000),
                Credito=c(1,0,1))
df4 <- data.frame(ID = c(2, 3), Publicidad = c(TRUE, FALSE))
df <- list(df1, df2, df3, df4)
reduce(df, full_join)
```

```
## Joining, by = "ID"
## Joining, by = "ID"
## Joining, by = "ID"
```



```
##   ID   Nombre Inicio_Compra Utlima_Compra   Pagos Credito Publicidad
## 1  1   Pepe        2011       2017-03-03    46690     1     NA
## 2  2   Juan        2017       2014-03-01    25456     0     TRUE
## 3  3   Luis        NA         <NA>          NA        NA    FALSE
## 4  4   Pedro       2000       2017-05-30    25000     1     NA
```

5 Actividades

1. Construye conjuntos de 10 números aleatorios que con media de -5,0,5 y 50.
2. Haciendo **babynames::births$births** se obtiene el número de nacimientos registrados en Estados Unidos desde 1909 hasta 2017. Usa las funciones de la librería purrr para determinar la media, la desviación estándar, el máximo, el mínimo y la mediana de los datos
3. Crea una función que muestre la media de cada columna de tipo numérica de un data frame
4. Construye una función parecida a summary, pero que opere únicamente sobre las columnas numéricas de un data frame.
5. Haciendo uso de los elementos aprendidos, simplifica la función **sum_neg_mejor** que desarrollamos en este capítulo. Comprueba tu resultado

Capítulo 14

Modelos Lineales

Capítulo 14 | Modelos Lineales

1 Introducción

Hasta ahora, la mayoría de los análisis que hemos realizado son descriptivos, es decir, nos sirven para conocer de un conjunto de datos, los estadísticos que mejor representan este conjunto. Podemos decir como se comportan, qué individuos (hogares o empresas, etc.) tienen mayores niveles de ingreso, o de gasto y como estos niveles cambian si se separa la información por género o edad. Es decir, hemos aprendido a identificar como las variables se relacionan entre si.

Sin embargo, si quisiéramos calcular, en promedio, cual seria el cambio en el ingreso si aumenta la educación de una persona, necesitamos hacer uso de otras herramientas, si bien pueden ser un poco más complejas, son también más interesantes y nos permiten establecer mejores relaciones entre las variables.

Si deseamos conocer los efectos del cambio de una variable sobre otra, necesitamos construir un modelo que nos permita identificar este efecto a manera que podamos describir una variable como función de otras. En estadística un modelo que sirve bastante para este problema es el modelo de regresión lineal. Anteriormente ya habíamos efectuado un modelo de regresión lineal, únicamente para ejemplificar el funcionamiento de la paquetería *purrr* por lo que no entramos en detalles. En este capítulo detallaremos los principios de la regresión lineal.

Si bien, este no es un curso de estadística, sería difícil no repasar algunos conceptos, después de todo, la estadística se encarga de obtener datos, organizar, resumir, presentar, analizar e interpretar esos datos para obtener conclusiones basadas en ellos.

El modelo de regresión lineal relaciona una variable (variable dependiente) como función de una o más variables (independientes) de forma lineal. El término lineal indica que el efecto que tienen las variables independientes sobre la variable dependiente es lineal. Esta técnica permite estudiar la dependencia que existen entre variables, es decir, una *dependencia estadística*.

La ecuación $y = b0 + b1x1 + u$ es una regresión lineal, mientras que la ecuación $y = b0 + b21\ x1 + u$ no es una regresión lineal.

La variable dependiente y esta explicada por la variable *x1*. La siguiente ecuación representa la regresión lineal entre ambas.

$$y = b0 + b1x1 + u$$

La siguiente ecuación representa la regresión lineal entre *y* y muchas variables independientes.

$$y = b0 + b1x1 + \ldots + b_K x_K + u$$

En las ecuaciones anteriores:

- *y* representa la variable dependiente, la variable que deseamos explicar
- $x1, \ldots, x_K$ representan las variables independientes
- *b0* es el intercepto o constante, el valor que toma la variable *y* cuando todas las variables independientes son iguales a cero
- *b1* es el coeficiente asociado a *x1, b2* corresponde a *x2*, etc.
- *u* es un término de error o residual que contendrá las diferencias entre lo que nuestro estima para la variable *y* y su valor real.

Dentro del análisis de regresión, los parámetros bk son los que necesitamos estimar y analizar.

Un punto importante que se debe aclarar, es que una relación estadística por sí misma no implica causalidad. La correlación entre dos variables indica el grado de asociación entre dos variables. Los coeficientes que se obtengan de una regresión indican el valor promedio de cambio en la variable dependiente cuando varia una variable independiente. Las relaciones estadísticas que se encuentren, si bien son útiles, no deben considerarse como relaciones deterministas.

Para este capítulo necesitamos instalar la librería *modelr* y *moments*, recordemos iniciar nuestro *script* de la manera ya acostumbrada: definiendo las librerías que necesitamos y definiremos el directorio para que mas adelante podamos cargar los datos de ejemplo que utilizaremos. Usaremos la librería **hexbin** que junto con **ggplot**, nos permitirá generar una gráfica con puntos exagonales. Finalmente la librería **splines2** nos permitirá construir modelos de orden superior.

```
setwd("~/Dropbox/Curso de R/Cap14_Cons_Modelos")
library (tidyverse)
library (modelr)
library (readr)
library (purrr)
library (lubridate)
library (tibble)
library (moments)
library(hexbin)
library(splines)
```

En este capítulo mostraremos los principios de la regresión lineal, si eres totalmente nuevo en este tema, es recomendable revisar antes algún libro de estadística para adquirir las nociones de este tema. No necesitas ser experto solo entender cuales son sus objetivos y tener claridad sobre sus supuestos.

2 Análisis de regresión

En este apartado estudiaremos el análisis de regresión y las herramientas que posee **R** para realizar esta tarea. Durante esta sección utilizaremos una base de datos simulada para entender cómo funciona y cómo validar una regresión lineal. Mas adelante daremos dos ejemplos usando bases de datos específicas.

Para construir un modelo simulado utilicemos los siguientes especificaciones:

```
# Construir datos
# Dos variables aleatorias
x1 ← rnorm(500,mean=10, sd=10)
x2 ← rnorm(500,mean=20, sd=20)
# Error con media 0 y var 1
u ← rnorm(500,mean=0, sd=1)
# b0=2, b1=5 y b2=10
y←2+(5*x1)+(10*x2)+u
datos ← tibble(y=y,x1=x1,x2=x2)
```

Observa que hemos establecido que el valor de la variable dependiente *y* es igual a 2 mas la multiplicación de la variable *x1* por 5, más 10 veces el valor de la variable *x2*. El objetivo del análisis de regresión es que dado un conjunto de valores de *y, x1, x2* poder encontrar los valores de ciertos parámetros (en esta caso 2, 5 y 10) que relacionan a y con *x1, x2.*

Antes de ejecutar la regresión, se asume que de manera previa ya realizamos un análisis descriptivo para identificar de un conjunto de datos cuales de nuestras variables se relacionan mejor con nuestra variable dependiente. Para efectuar este análisis previo, debiste haber usado todas las herramientas que ya hemos aprendido.

Para ejecutar la regresión linean utilizamos la función *lm()*, la sintaxis es la siguiente; primero indicar la variable dependiente, luego el signo ~ y enseguida las variables independientes separadas por el signo +. Con la opción *data*= se debe especificar el *data frame* (conjutno de datos) donde se encuentren las variables necesarias para el análisis:

```
reg ← lm(y ~ x1+x2, data=datos)
summary(reg)
```

```
##
## Call:
## lm(formula = y ~ x1 + x2, data = datos)
##
```



```
## Residuals:
##    Min        1Q        Median    3Q       Max
## -2.8935    -0.6637    -0.0104    0.5961   3.9762
##
## Coefficients:
##              Estimate Std. Error t value   Pr(>|t|)
## (Intercept)  2.047985  0.079400   25.79     <2e-16 ***
## x1           5.005880  0.004526   1106.05   <2e-16 ***
## x2           9.997524  0.002249   4446.16   <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error:   0.9794 on 497 degrees of freedom
## Multiple R-squared:        1,     Adjusted R-squared: 1
## F-statistic: 1.013e+07 on 2 and 497 DF,  p-value: < 2.2e-16
```

Se recomienda que al realizar una regresión se asignen los resultados a un objeto, pues toda la información necesaria para continuar con el análisis se guardara en ese objeto. Además, si no se asigna a un objeto y realiza una regresión, el comando *lm()* de manera predeterminada solo mostrara los coeficientes estimados. Para obtener más detalles de los resultados ejecutamos la instrucción *summary(reg)*. Con ello obtenemos la siguiente información:

- Medidas de posición de los residuos
- El intercepto y los coeficientes de las variables independientes
- Los errores estándar, estadístico *t* y valores *p* de los coeficientes
- Estadístico *F* y su valor *p*
- Coeficiente de determinación ó r^2 y la r^2 ajustada

El modelo ha estimado que la relación entre *x1,x2* con la variable *y* esta dada por los coeficientes en la columna de nombre *Estimated*. Se observa que los coeficientes estimados son cercanos a los verdaderos y son estadísticamente significativos (importantes para predecir el valor de y). Esto lo sabemos porque los valores de *Pr(>|t|)* son extremadamente pequeños. *R* ayuda a determinar la significancia estadística con un código de asteriscos el cual se incluye al final como

```
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

De todos los elementos que muestran al efectuar *summary(reg)* los de mayor relevancia serán ade-

mas de los coeficientes estimados y de *Pr(>|t|)* (comocido como p-value) el valor *Multiple R-squared*, el cual toma valores entre 0 y 1, indicando el porcentaje de los cambios en y que pueden ser explicados por los cambios en la variables *x1,x2*. Obviamente entre mayor sea este valor, mejor será modelo. Si estos términos te suenan

totalmente desconocidos te recomendamos revisar algún libro de estadística para familiarizarte un poco con la idea de la regresión lineal.

2.1 Variables categóricas

Dentro del análisis de regresión es posible añadir variables categóricas Recordemos que este tipo de variables se refieren a características de los individuos, como el sexo, la religión, la preferencia sexual, la entidad de residencia, etc. Para añadir este tipo de variables al análisis se debe hacer en forma de variables binarias y son de manera comparativa.

Consideremos el ejemplo del sexo, supongamos que la base de datos simulada es sobre individuos y deseamos conocer si el género tiene un impacto sobre la variable dependiente y. Para añadir esta variable se debe construir una variable dicotómica, que tome el valor de 0 si el individuo es hombre y 1 si el individuo es mujer. De esta manera la regresión captura los efectos que tiene esta variable cuando el valor de 1. No se debe de incluir una variable binaria para hombres y otra para mujeres. El valor del coeficiente asociado a esta variable binaria solo existe cuando es 1.

Continuando con el ejercicio simulado, construiremos ahora una variable para añadir la variable categórica entre cero y uno, y construimos de nuevo la variable dependiente *y*

```
genero <- round(runif(500,min=0,max=1))
y<-2+(5*x1)+(10*x2)+(5*genero)+u
datos <- tibble(y=y,x1=x1,x2=x2,sexo=genero)
```

round(runif(500,min=0,max=1)) nos permite generar 500 valores entre 0,1 mientras. Estos valores se redondearán con la función **round**. Observa que genero, sólo contiene 0 y 1.

```
genero
## [1]   0 0 1 1 0 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1
0 1
## [38]  1 0 0 0 1 0 1 1 0 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 1
1 1
## [75]  0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 1 0 0 0 1
0 0
## [112] 0 0 1 0 1 1 0 1 0 0 1 0 0 1 0 1 0 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1 0 1 1
1 0
## [149] 0 0 0 1 1 1 1 0 0 1 0 1 1 1 0 0 1 1 1 1 0 1 0 1 1 1 0 1 1 0 0 0 1 1 1
1 1
```



```
## [186] 0 0 0 1 1 1 0 1 1 1 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 0 1 1 1 1 1
0 0
## [223] 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 1 0 1 0 0 1 1 0 0 1 0 0 1 1 1 1 1 0 0 1
1 0
## [260] 1 0 1 0 0 0 1 1 1 0 1 1 0 1 1 0 0 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 0
0 0
## [297] 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 0 1 0 1 0 1 0 0 1 1 0 0 1 1 0 0
0 1
## [334] 0 0 1 0 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 1 1 0 1 1 0 0 1 1 1 0 1 1 0
0 1
## [371] 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 0 1 1 0 1 0 0 1 0 0 0 0 1 0 0 1 0
1 1
## [408] 1 1 0 1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 1 1 1
0 0
## [445] 1 1 0 0 1 0 1 1 1 1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1 1 1 0 1 0 0 1 1 1
1 1
## [482] 1 1 0 0 0 0 1 0 1 1 1 0 1 1 1 1 1 0 1
```

Estos no permitirá ver si hay diferencias en el impacto de *x1,x2* sobre y cuando la variable sexo vale 1, que cuando vale 0.

Con esta modificación generamos nuevamente el modelo. Para estimar su efecto ahora añadimos a la ecuación a la variable sexo:

```
reg2 <- lm( y ~ x1+x2+sexo, data=datos)
summary(reg2)
##
## Call:
## lm(formula = y ~ x1 + x2 + sexo, data = datos)
##
## Residuals:
##    Min        1Q    Median     3Q    Max
##    -2.8977 -0.6603 -0.0143 0.5947 3.9721
##
## Coefficients:
##                Estimate Std.    Error      t value    Pr(>|t|)
## (Intercept)    2.043228         0.091082   22.43      <2e-16 ***
## x1             5.005838         0.004548   1100.74    <2e-16 ***
```

```
## x2           9.997538        0.002254   4435.19     <2e-16 ***
## sexo         5.009438        0.088251   56.76       <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9804 on 496 degrees of freedom
## Multiple R-squared: 1, Adjusted R-squared: 1
## F-statistic: 6.736e+06 on 3 and 496 DF, p-value: < 2.2e-16
```

El coeficiente asociado a esta variable es de 5.0094377, es decir, el valor de y aumenta en 5.0094377 si el individuo es mujer (la variable sexo vale 1). Otra interpretación seria, las mujeres, en promedio, tienen un valor y más alto que los hombres en 5.0094377. Este tipo de análisis es muy común cuando se estudian por ejemplo diferencias en el salario o el color de piel de una persona y su impacto en su sueldo. Debido a que el *p-value* es casi cero, podemos concluir que si hay una diferencia en el sexo para explicar la variable *y*. Si por ejemplo esta valor *p-vale* hubiera resultado mayor a 0.05, indicaría que el sexo no explica las diferencias en la variable *y*.

Naturalmente, **R** nos permite escribir en la ecuación de la regresión una variable de tipo factor, y de manera automática creara las variables binarias. Consideremos el siguiente ejemplo, transformando la variable sexo a una variable de tipo factor:

```
datos <- datos %>%
     mutate(sexo2=as.character(sexo)) %>%
     mutate(sexo2 = fct_recode(sexo2,
"Hombre" = "0",
"Mujer" = "1"))
```

Ahora estimamos la regresión:

```
reg3 <- lm( y ~ x1+x2+sexo2, data=datos)
summary(reg3)
##
## Call:
## lm(formula = y ~ x1 + x2 + sexo2, data = datos)
##
## Residuals:
##    Min   1Q        Median     3Q        Max
## -2.8977 -0.6603    -0.0143    0.5947    3.9721
```



```
##
## Coefficients:
##               Estimate Std. Error   t value    Pr(>|t|)
## (Intercept)   2.043228   0.091082   22.43      <2e-16 ***
## x1            5.005838   0.004548   1100.74    <2e-16 ***
## x2            9.997538   0.002254   4435.19    <2e-16 ***
## sexo2Mujer    5.009438   0.088251   56.76      <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9804 on 496 degrees of freedom
## Multiple R-squared: 1, Adjusted R-squared: 1
## F-statistic: 6.736e+06 on 3 and 496 DF, p-value: < 2.2e-16
```

Donde obtenemos el mismo resultado, sólo que ahora es mas fácil identificar la variable sexo y su relación cuanto se trata de una mujer.

Para observar la matriz de datos que utiliza **R** para estimar la regresión lineal, podemos utilizar la función *model_matrizx()*, la cual requiere que indiquemos el *data frame,* seguido de la ecuación que deseamos estimar:

```
model_matrix(datos, y ~ x1+x2+sexo2)
## # A tibble: 500 x 4
##    `(Intercept)`   x1         x2         sexo2Mujer
##          <dbl>     <dbl>      <dbl>      <dbl>
## 1        1         14.0       6.16       0
## 2        1         12.3       -12.8      0
## 3        1         -9.82      8.28       1
## 4        1         11.7       -0.203     1
## 5        1         38.1       -1.89      0
## 6        1         5.58       24.4       1
## 7        1         2.16       -3.65      1
## 8        1         6.70       29.9       1
## 9        1         10.1       2.19       1
## 10       1         -0.00797   10.5       1
## # ... with 490 more rows
```

Notesé que aparece una columna de unos, el cual se refiere al intercepto, y la variable categórica sexo, ahora aparece como una variable binaria.

Nota que en los resultados aparece la variable *sexo2Mujer* indicado que es una variable binaria, que toma el valor de 1 cuando *sexo2* es igual a *Mujer* y 0 cuando es igual a *Hombre.*

Si tenemos más de dos categorías, entonces serian necesarias dos variables binarias. En general, si tenemos una variable con *k* categorías, se deben incluir *k-1* variables binarias. La categoría que sea omitida es la base con la cual se comparan las demás variables.

Consideremos el siguiente ejemplo, crearemos una variable con 4 categorías:

```
c ← c("a","b","c","d")
datos ← tibble(datos, categoria = rep(c,125))
```

Estimamos la regresión:

```
reg4 ← lm(y ~ x1+x2+categoria, data=datos)
summary(reg4)
```

```
##
## Call:
## lm(formula = y ~ x1 + x2 + categoria, data = datos)
##
## Residuals:
##    Min       1Q       Median    3Q       Max
## -5.0434    -2.4987   0.4766    2.3921   5.9468
##
## Coefficients:
##              Estimate Std.  Error     t value    Pr(>|t|)
## (Intercept)  4.502020       0.292996  15.365     <2e-16 ***
## x1           5.028523       0.012420  404.861    <2e-16 ***
## x2           9.990168       0.006202  1610.681   <2e-16 ***
## categoriab   0.274442       0.341653  0.803      0.422
## categoriac   0.039091       0.340755  0.115      0.909
## categoriad   -0.020349      0.340639  -0.060     0.952
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```



```
## Residual standard error: 2.687 on 494 degrees of freedom
## Multiple R-squared: 0.9998, Adjusted R-squared: 0.9998
## F-statistic: 5.379e+05 on 5 and 494 DF, p-value: < 2.2e-16
```

Note que aparecen variables para las categorías *b*, *c* y *d*, por lo tanto la base es la categoría *a*.

Los resultados de la regresión *4*, nos indican que las variables *categoriab, categoriac, categoriad* no son significativas (no son relevantes para predecir el valor de *y*). Esto lo sabemos porque los valores correspondientes a ellas en la columna *Pr(>|t|)* son mayores a 0.05. Esta es una regla usa en estadística para determinar la significancia de una variable, la razón de ello esta muy fuera del objetivo de este capítulo.

2.2 Interacción

La interacción entre dos variables se refiere a una situación en la que su variación depende de una tercera variable. En este apartado te presentamos los elementos para que puedas ejecutar una regresión lineal con interacciones en *R*. No entraremos en detalles estadísticos sobre estas relaciones, únicamente nos enfocaremos en su uso en el entorno de programación.

Para este ejercicio consideremos los siguientes datos creados anteriormente. Para ver la relación que hay entre las variables. Existen dos modelos posibles, uno que considere la interacción entre ellas multiplicadas y el otro sumadas. Esto sucede porque una de las variables toma valores categóricos, es decir la variable x2, toma valores de a,b,c o d. Definamos ambos modelos:

```
reg5 ← lm(y ~ x1 + x2 + sexo + categoria, datos)
reg6 ← lm(y ~ x1 + x2 + sexo*categoria, datos)
summary(reg5)
```

```
##
## Call:
## lm(formula = y ~ x1 + x2 + sexo + categoria, data = datos)
##
## Residuals:
##    Min       1Q       Median    3Q       Max
##    -2.9401   -0.6570  -0.0122   0.5785   3.9250
##
## Coefficients:
##              Estimate Std. Error   t value    Pr(>|t|)
## (Intercept)  1.987913   0.115924   17.148     <2e-16 ***
## x1           5.005876   0.004557   1098.534   <2e-16 ***
## x2           9.997348   0.002270   4403.425   <2e-16 ***
```

```
## sexo          5.009621   0.088483   56.617     <2e-16 ***
## categoriab    0.103657   0.124901   0.830      0.407
## categoriac    0.016747   0.124537   0.134      0.893
## categoriad    0.114753   0.124517   0.922      0.357
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.982 on 493 degrees of freedom
## Multiple R-squared:      1, Adjusted R-squared: 1
## F-statistic: 3.357e+06 on 6 and 493 DF, p-value: < 2.2e-16
```

```
summary(reg6)
```

```
##
## Call:
## lm(formula = y ~ x1 + x2 + sexo * categoria, data = datos)
##
## Residuals:
##    Min        1Q         Median     3Q         Max
##    -2.9255    -0.6747    -0.0060    0.5808     3.9689
##
## Coefficients:
##                     Estimate Std.  Error      t value    Pr(>|t|)
## (Intercept)         2.023762       0.140342   14.420     <2e-16 ***
## x1                  5.005493       0.004579   1093.091   <2e-16 ***
## x2                  9.997495       0.002285   4375.257   <2e-16 ***
## sexo                4.942756       0.176100   28.068     <2e-16 ***
## categoriab          0.114612       0.182286   0.629      0.530
## categoriac          -0.117939      0.180371   -0.654     0.514
## categoriad          0.096408       0.177564   0.543      0.587
## sexo:categoriab     -0.019029      0.249467   -0.076     0.939
## sexo:categoriac     0.258091       0.249873   1.033      0.302
## sexo:categoriad     0.032138       0.249591   0.129      0.898
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



```
##
## Residual standard error: 0.9834 on 490 degrees of freedom
## Multiple R-squared: 1, Adjusted R-squared: 1
## F-statistic: 2.231e+06 on 9 and 490 DF, p-value: < 2.2e-16
```

En el caso del modelo que efectúa la interacción multiplicada, ademas de presentar los efectos en el cambio que presenta el modelo de suma, incluye el cambio en y cuando cambia la interacción entre x1 y las diferentes categorías.

En el modelo que se encuentra sumado, obtenemos un coeficiente x1, que indica como cambia y cuando cambia x1. El resto de los coeficientes indica el cambio de y cuando la observación pertenece a cada una de las diferentes categorías. En el caso del modelo que efectúa la interacción multiplicada, ademas de presentar los efectos en el cambio que presenta el modelo de suma, incluye el cambio en y cuando el elemento de la observación pertenece tanto una determinada categoría como a un determinado sexo.

Los principios que hemos establecido detallan la regresión lineal. Recuerda que el objetivo es encontrar los coeficientes que multiplicados con las variables $x's$ generen a la variable y . Estos modelos permiten generar una predicción para el comportamiento de y. La diferencia entre el valor que el modelo predice para un determinado y y su valor real, es el error del modelo. Con el fin de analizar los errores en la estimación de estas variables, generemos la predicción para la variable y que efectúa cada uno de ellos.

3 Ajuste del modelo

Una vez realizado el análisis de regresión debemos revisar el ajuste del modelo. Es decir, necesitamos tener una medida para poder determinar el modelo de regresión que estimamos es bueno o no para obtener conclusiones de los datos. Para este punto es necesario mencionar algunos de los supuestos básicos de la regresión lineal y que sucede si no se cumplen. Explorar a detalle es una tarea que esta fuera de los objetivos de este curso, sin embargo, deben tenerse en cuenta que según los objetivos del análisis realizado se puede ser más o menos flexible con estos supuestos.

El análisis de regresión considera, entre otros, los siguientes supuestos:

1 El valor esperado de los residuos es cero 2 El error esta normalmente distribuido 3 La covarianza entre los errores y las variables independientes es cero 5 La varianza del los residuos es constante 4 No existe colinealidad entre las variables independientes

Los supuestos 1 y 2, no son un problema para muestras grandes. Ver a detalle estos supuestos y las maneras *formales* de detectar y corregir estas situaciones nos alejaría mucho del tema, por lo que si el usuario no esta familiarizado con la teoría de la regresión lineal se recomienda revisarla. En su caso, a continuación mostraremos algunas formas o reglas practicas para detectar estos problemas y determinar si el modelo a estimar más o menos bueno. Los problemas 1 a 4 requiere analizar los residuos mientras que el punto 5 trata sobre las variables independientes.

3.1 Predecir valores de la regresión

Antes de continuar, limpiemos nuestros ambiente de trabajo, para ello usaremos remove y generemos nuevamente la regresión de ejemplo básico con el que comenzamos.

```
remove(reg, reg2, reg3, reg4, reg5, reg6)
reg ← lm( y ~ x1+x2+sexo, data=datos)
```

Con el fin de analizar los errores en la estimación de estas variables, generemos los residuos de la regresión *reg*. Esto se obtiene de dos formas, con el uso de la función **add_predictions** para que obtengamos la estimación de la variable dependiente, la cual aparecerá con el nombre *pred*, para añadir los residuos, los cuales aparecen con el nombre *resid,* utilizamos la función **add_residuals**, ambas funciones forman parte de la librería *modelr*:

```
predict← datos %>%
add_predictions(reg) %>%
add_residuals(reg)

predict
```

```
## # A tibble: 500 x 8
##    y        x1        x2       sexo  sexo2 categoria pred  resid
##    <dbl>    <dbl>     <dbl>    <dbl> <fct> <chr>     <dbl> <dbl>
## 1  133.     14.0      6.16     0     Hombre a        134.  -1.02
## 2  -64.7    12.3      -12.8    0     Hombre b        -64.5 -0.169
## 3  40.0     -9.82     8.28     1     Mujer  c        40.7  -0.722
## 4  63.3     11.7      -0.203   1     Mujer  d        63.4  -0.0806
## 5  174.     38.1      -1.89    0     Hombre a        174.  0.484
## 6  277.     5.58      24.4     1     Mujer  b        279.  -1.30
## 7  -16.6    2.16      -3.65    1     Mujer  c        -18.7 2.09
## 8  339.     6.70      29.9     1     Mujer  d        339.  -0.192
## 9  78.2     10.1      2.19     1     Mujer  a        79.3  -1.11
## 10 112.     -0.00797  10.5     1     Mujer  b        112.  0.487
## # ... with 490 more rows
```

Observa que el objeto *predict* contiene los datos originales, mas las columnas mencionadas. *pred* contienen la predicción de *y* y *resid* la diferencia entre la predicción y el valor real.

A continuación mostramos algunos puntos deben tomarse en cuenta al estimar un modelo de regresión lineal. Generalmente es más sencillo detectar estos problemas que corregirlos, y para

esto último necesitaríamos adentrarnos a otro tipo de herramientas de análisis y desviarnos del tema, pero lo mencionamos para no perder de vista que un modelo de regresión necesita tener cierto grado de certidumbre.

3.1.1 Residuos con media cero y normales

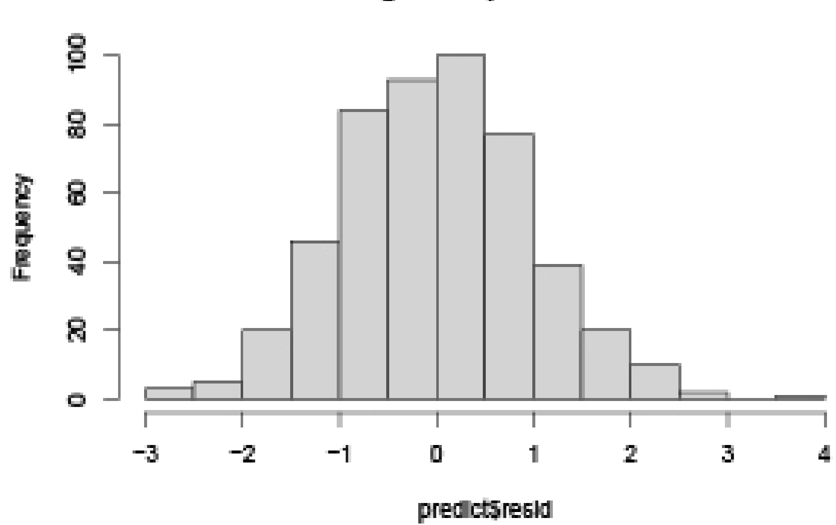
En nuestro conjunto de datos, ya tenemos los residuos que nuestra regresión genera, el primera análisis de los residuos consiste en ver la covarianza de los errores para saber si las variables son independientes. Esto lo podemos hacer de dos formas, la primera con un diagrama de dispersión o a través del calculo de la covarianza, así como el promedio de los residuos:

```
predict %>%
  summarise(
     media_ui = mean(resid),
     asimetria_ui = skewness(resid),
     kurtosis_ui = kurtosis(resid),
     cov_x1_ui = cov(x1, resid),
     cov_x2_ui = cov(x2, resid),
     cov_sexo_ui = cov(sexo, resid)
)
```

```
## # A tibble: 1 x 6
##    media_ui asimetria_ui kurtosis_ui cov_x1_ui cov_x2_ui cov_sexo_ui
##    <dbl>    <dbl>        <dbl>       <dbl>     <dbl>     <dbl>
## 1  1.08e-13 0.159        3.40        -7.82e-14 1.29e-13  -4.45e-15
```

```
hist(predict$resid)
```





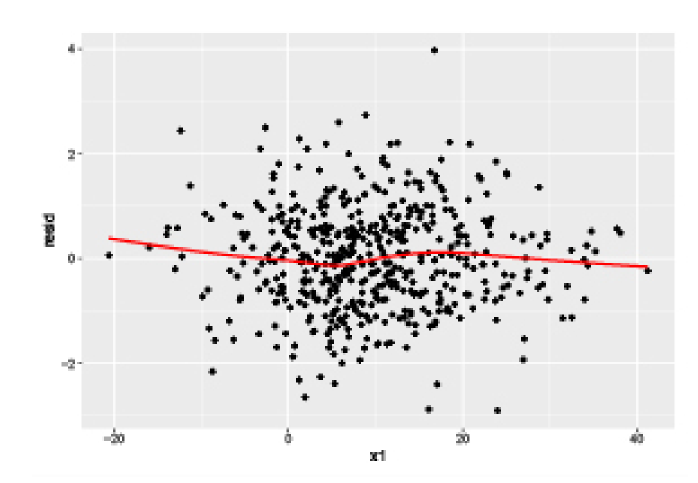
Hemos solicitado un conjunto de medidas estadísticas para analizar la relación entre los residuos (errores del modelo) y las variables explicativas.

Se verifica además que no esta correlacionados pues **cov_x1_ui, cov_x2_ui cov_sexo_ui** tienen valores cercanos a cero. Ademas los residuales (contenidos en la variable resid) tienen una forma mas o menos acampanada.

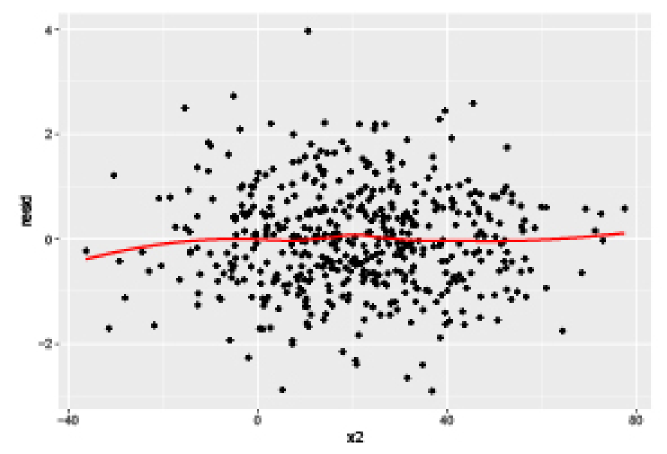
El problema de los residuos con media cero y distribuidos normalmente pierde relevancia si la muestra es grande. Sin embargo, si la muestra es pequeña, es importante que se cumpla este supuesto.

Para analizar la relación usando un diagrama de dispersión, hacemos;

```
predict %>%
     ggplot(aes(x = x1, y = resid)) +
     geom_point()+
     geom_point() +
geom_smooth(color = "red", se = FALSE)
```




```
predict %>%
     ggplot(aes(x = x2, y = resid)) +
     geom_point()+
     geom_point() +
geom_smooth(color = "red", se = FALSE)
```

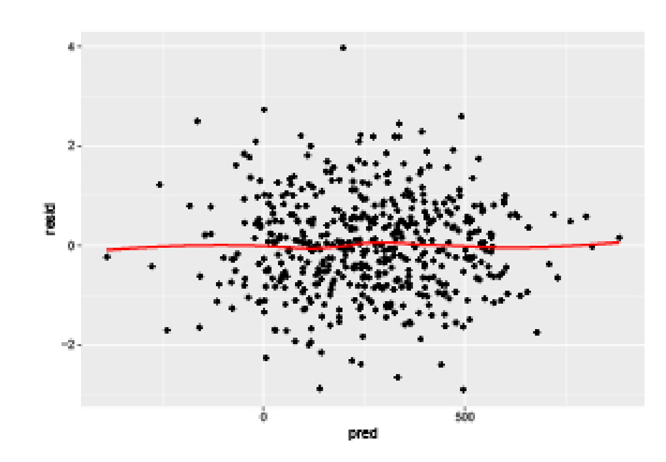


3.1.2 Residuos con varianza constante

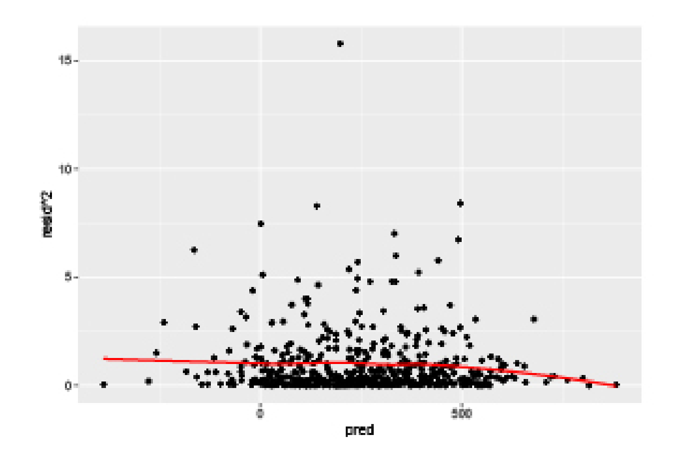
Una forma gráfica de verificar si la varianza del modelo es constante para los diferentes valores de las variables independientes es graficar los residuos al cuadrado contra la variable dependiente estimada.

Los residuales deben de distribuirse de forma aleatoria. Si se observa algún patrón o formas de cono, o más dispersión en los extremos, son indicadores de que los residuos no tienen varianza constante.

```
predict %>%
     ggplot(aes(x = pred, y = resid)) +
     geom_point()+
     geom_point() +
geom_smooth(color = "red", se = FALSE)
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
predict %>%
     ggplot(aes(x = pred, y = resid^2)) +
     geom_point()+
geom_point() +
geom_smooth(color = "red", se = FALSE)
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Grafica como estas son las que nos indican que las residuos tienen variable constante, lo cual es un punto a favor del modelo.

Si se detecta que la varianza de los residuos no es constante una posible solución es realizar una transformación de los todos, por ejemplo, estimar la regresión lineal sobre el logaritmo natural de *y*, contra los logaritmos naturales de *x1 y x2.* También una de las fuentes de este problema son los datos atípicos, por lo se recomienda revisar la existencia de estos.

3.1.3 Colinealidad

La colinealidad se refiere a que las variables independientes están correlacionadas. Algunas de las causas de este problema pueden ser la mala especificación del modelo, como sería incluir términos polinomiales a la regresión. Un modelo sobredeterminado también puede ser la causa de este problema, es decir, se incluyen más variables independientes que el número de observaciones. Este problema también puede surgir si una variable independiente es una transformación lineal de otra variable independiente.

Para detectar si nuestro modelo tiene algún problema de colinealidad, podemos verificar los siguientes puntos:

- Correlación entre las variables independientes
- Alto valor del *R-squared* pero pocas variables significativas

```
cor(datos[,2:4])
```

```
##               x1          x2        sexo
## x1     1.00000000 -0.09091892  0.09224441
## x2    -0.09091892  1.00000000 -0.06225200
## sexo   0.09224441 -0.06225200  1.00000000
```

Esto nos indica la correlación entre las variables,

Si detectamos estos problemas algunas soluciones son:

- eliminar variables
- transformar variables
- aumentar el tamaño de muestra

4 Ejemplo 1: Enigh

Ahora que hemos analizado en términos generales, en que consiste la regresión lineal, estimaremos un modelo utilizando datos reales de la encuesta ENIGH. Supongamos que nos interesa estimar un modelo que nos permita identificar en promedio cuanto aumenta el gasto en alimentos, si el tamaño del hogar crece.

Anteriormente trabajamos con una muestra de la encuesta ENIGH de 2018, tomemos de nuevo esta base de datos. Antes de comenzar, limpiemos completamente nuestro ambiente de trabajo. Eso lo podemos ejecutar con la función **rm(list=ls())**

```
# Limpiar el ambiente de trabajo
rm(list=ls())
# Cargar datos
enigh <- read_csv("hogares_enigh.csv")
head(enigh)
```

```
## # A tibble: 6 x 26
## folioviv foliohog ubica_geo tam_loc est_socio est_dis upm factor clase_hog
##   <chr>      <dbl> <chr>       <dbl> <dbl> <chr>     <chr> <dbl>  <dbl>
## 1 0100013~   1     01001       1     3     002       0000~  175       2
## 2 0100013~   1     01001       1     3     002       0000~  175       2
## 3 0100013~   1     01001       1     3     002       0000~  175       2
## 4 0100013~   1     01001       1     3     002       0000~  175       2
## 5 0100013~   1     01001       1     3     002       0000~  175       2
## 6 0100026~   1     01001       1     3     002       0000~  189       2
## # ... with 17 more variables: sexo_jefe <dbl>, edad_jefe <dbl>,
## # educa_jefe <chr>, tot_integ <dbl>, percep_ing <dbl>, ing_cor <dbl>,
## # ingtrab <dbl>, gasto_mon <dbl>, alimentos <dbl>, vesti_calz <dbl>,
## # vivienda <dbl>, limpieza <dbl>, salud <dbl>, transporte <dbl>,
## # educa_espa <dbl>, personales <dbl>, transf_gas <dbl>
```

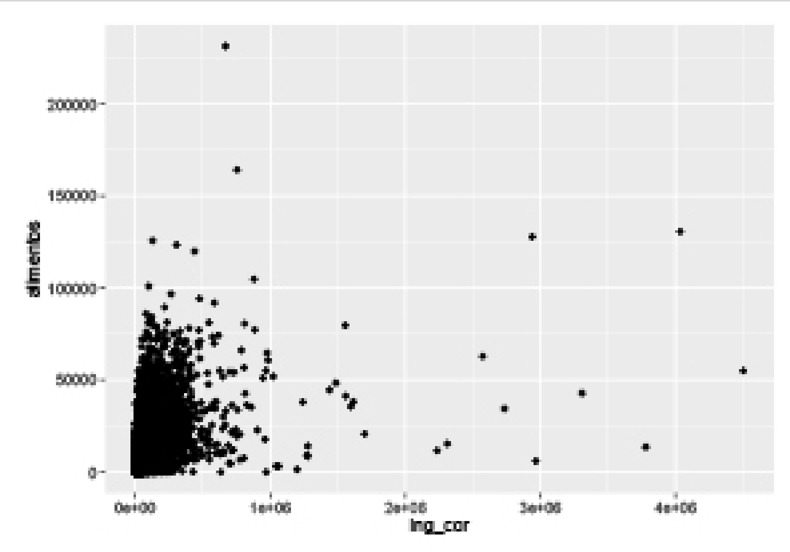
En la muestra que tenemos observaremos la variable *alimentos*, que representa el gasto trimestral del hogar por este concepto, la variable *ing_cor* que representa el ingreso trimestral del hogar. Dado que ya exploramos un poco esta encuesta, realicemos el siguiente cambio en algunas variables categóricas; *sexo_jefe, clase_hog* y *educa_jefe.*

```
# Renombrar Factores
enigh <- enigh %>%
     mutate(sexo_jefe=as.character(sexo_jefe)) %>%
     mutate(clase_hog=as.character(clase_hog)) %>%
     mutate(sexo_jefe = fct_recode(sexo_jefe, "Hombre" = "1", "Mujer" = "2")) %>%
     mutate(clase_hog = fct_recode(clase_hog,
                "Unipersonal" = "1",
                "Nuclear" = "2",
                "Ampliado" = "3",
                "Compuesto" = "4",
```

```
                 "Corresidente" = "5")) %>%
mutate(educa_jefe = fct_recode(educa_jefe,
                 "Sin instrucción" = "01",
                 "Preescolar" = "02",
                 "Primaria incompleta" = "03",
                 "Primaria completa" = "04",
                 "Secundaria incompleta" = "05",
                 "Secundaria completa" = "06",
                 "Preparatoria incompleta" = "07",
                 "Preparatoria completa" = "08",
                 "Profesional incompleta" = "09",
                 "Presional completa" = "10",
                 "Posgrado" = "11"))
```

Analicemos la relación existente, entre el ingreso monetario del hogar y el gasto en alimentos, un gráfico de dispersión nos puede ayudar a entender la relación entre ambas variables. Con el siguiente gráfico se puede observar que existen valores extremos. En el capítulo tres, ya analizamos como mejorar la visualización si se presenta este tipo de casos.

```
enigh %>%
      ggplot(aes(x = ing_cor, y = alimentos)) +
      geom_point()
```



Evidentemente, existen valores atípicos que pueden sesgar nuestras variables. Algunas variables económicas después de una transformación logarítmica tiene una distribución normal. Este tipo de transformacionespuede mejorar el modelo y en ocasiones facilitar la interpretación de los resultados. Calculemos el logaritmo natural de nuestras variables de interés gasto en alimentos e ingreso del hogar.

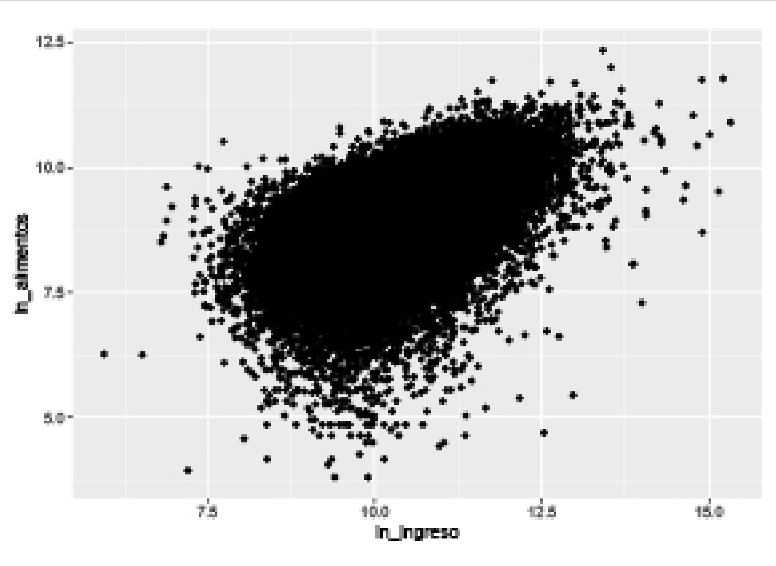
Para evitar errores, transformaremos los valores cero a valores perdidos y las filtraremos. Esta limpieza de datos, también en capítulos anteriores.


```
# Para evitar errores, transformaremos los valores cero a valores perdidos y
las filtraremos:
enigh <- enigh %>%
      mutate(ing_cor = ifelse(ing_cor==0, NA, ing_cor)) %>%
      mutate(alimentos = ifelse(alimentos==0, NA, alimentos)) %>%
      filter(ing_cor!="NA") %>%
      filter(alimentos!="NA")

# Creamos en logaritmo natural
enigh <- enigh %>%
      mutate(ln_alimentos=log(alimentos)) %>%
      mutate(ln_ingreso=log(ing_cor))
```

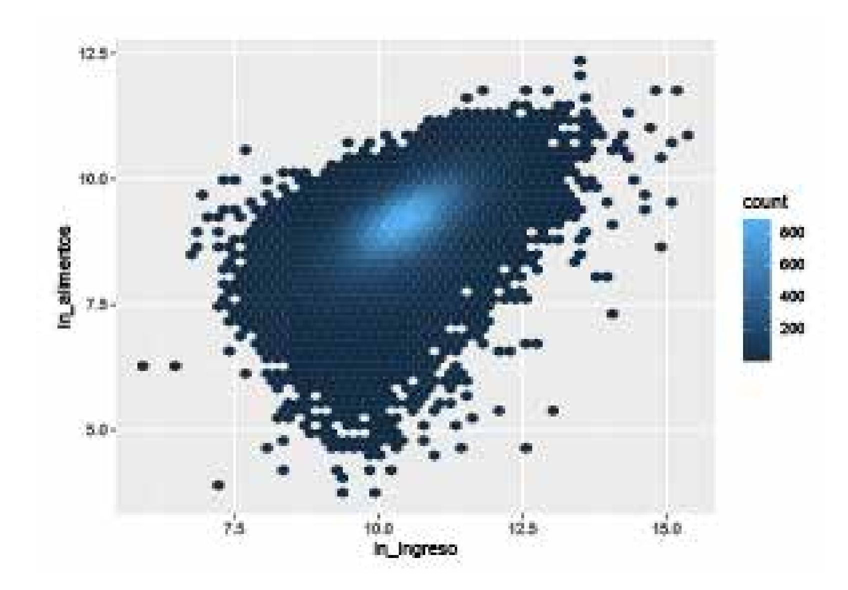
Si analizamos la dispersión entre el logaritmo natural del ingreso y del gasto en alimentos, se observa una relación positiva más clara:

```
enigh %>%
      ggplot(aes(x = ln_ingreso, y = ln_alimentos)) +
      geom_point()
```

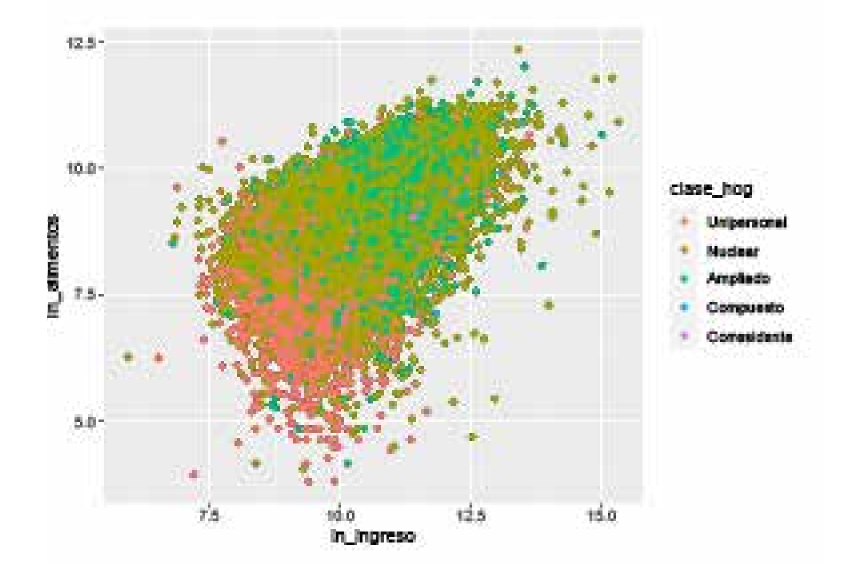


Otra forma de visualizar esta información es un gráfico como el siguiente, donde el color indica donde se concentran los hogares, o también colorear los puntos según alguna variable categórica, para identificar si existen algunas diferencias entre grupos:

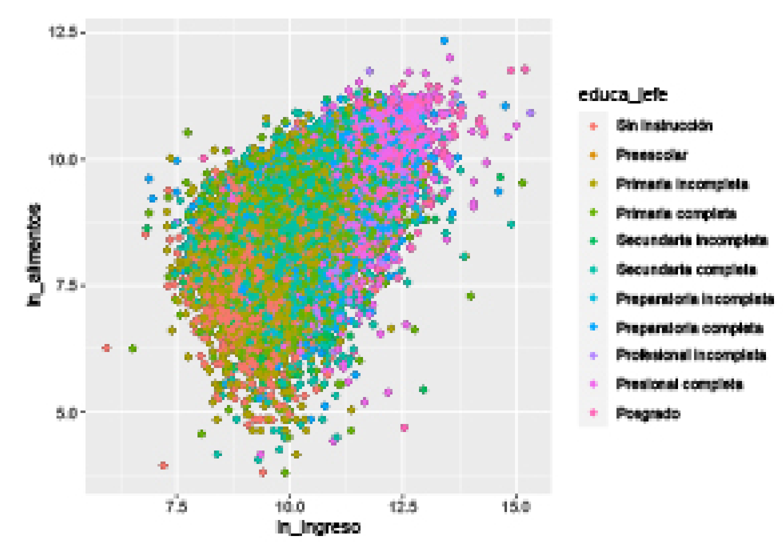
```
ggplot(enigh, aes(ln_ingreso, ln_alimentos)) +
geom_hex(bins = 50)
```



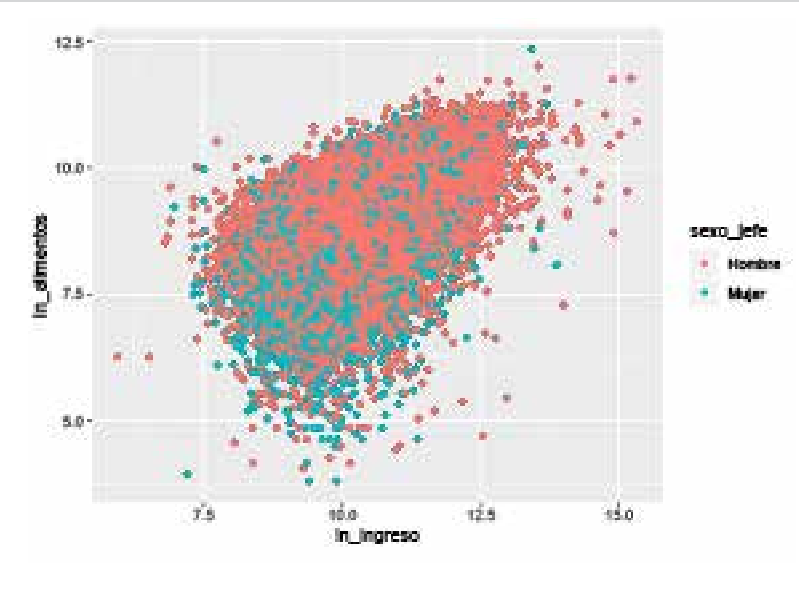

```
enigh %>%
     ggplot(aes(x = ln_ingreso, y = ln_alimentos)) +
     geom_point()+
     geom_point(aes(color = clase_hog))
```




```
enigh %>%
     ggplot(aes(x = ln_ingreso, y = ln_alimentos)) +
     geom_point()+
     geom_point(aes(color = educa_jefe))
```




```
enigh %>%
     ggplot(aes(x = ln_ingreso, y = ln_alimentos)) +
     geom_point()+
     geom_point(aes(color = sexo_jefe))
```

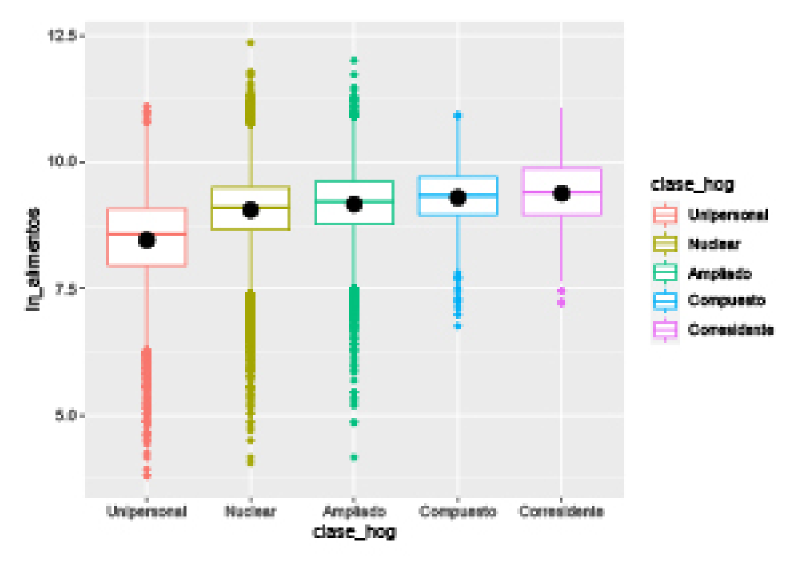


Con esta información podemos decir, que el gasto en alimentos, depende del tamaño del hogar, y del tipo de hogar. Los hogares conformados por una sola persona evidentemente destinan menos gasto a este concepto. En cuanto a la educación, los hogares donde el jefe el hogar tiene un mayor escolaridad, indican que en promedio tienen mayor gasto en este rubro.

Generemos los diagramas de caja para ver como es el gasto del hogar en relación a los diferentes tipos de hogar y niveles educativos.

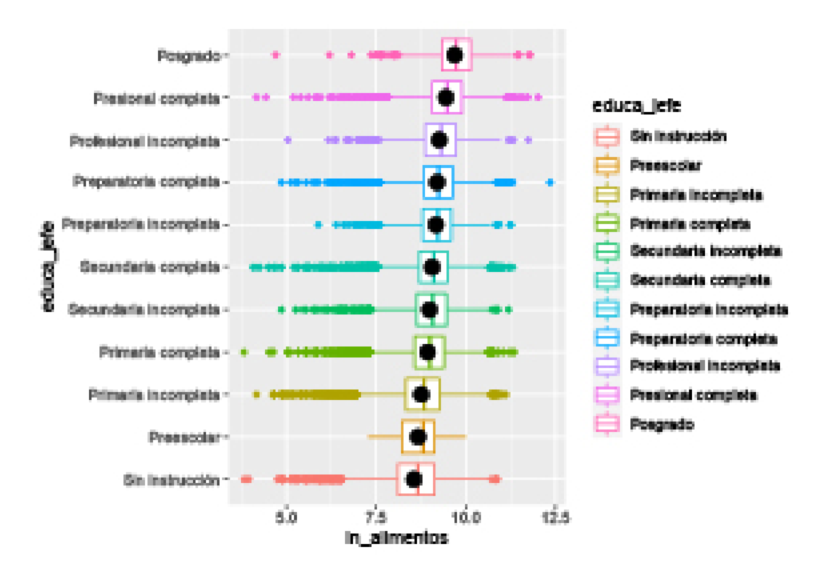
```
sumg← enigh %>%
     group_by(clase_hog) %>%
     summarise(ln_alimentos=mean(ln_alimentos))

ggplot(enigh,aes(clase_hog, ln_alimentos, color=clase_hog)) +
     geom_boxplot() +
     geom_point(data=sumg, color="black", size=4)
```

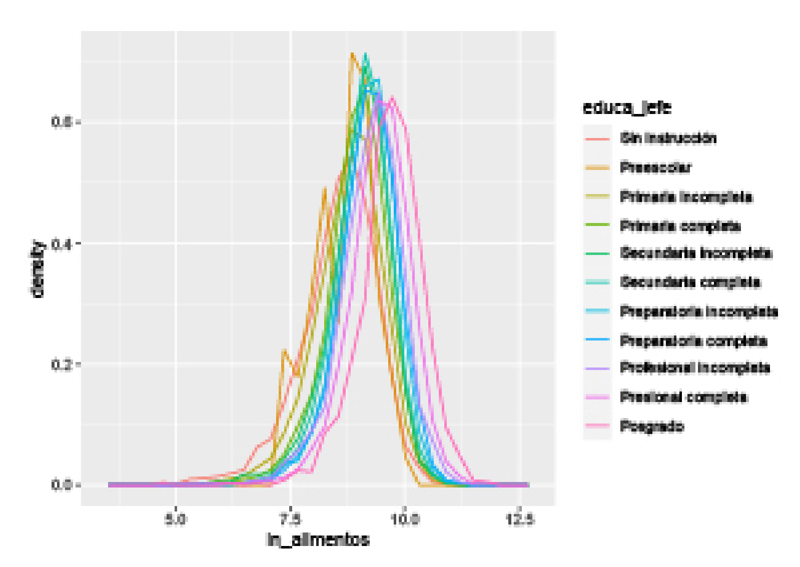



```
sumg← enigh %>%
     group_by(educa_jefe) %>%
     summarise(ln_alimentos=mean(ln_alimentos))

ggplot(enigh, aes(educa_jefe, ln_alimentos, color = educa_jefe)) +
     geom_boxplot()+
     geom_point(data=sumg, color="black", size=4) +
     coord_flip()
```




```
ggplot(
data = enigh,
mapping = aes(x = ln_alimentos, y = ..density..)
) +
geom_freqpoly(mapping = aes(color = educa_jefe))
```



Para determinar el efecto sobre el gasto en alimentos, si aumenta el tamaño del hogar en una persona, o la edad del jefe del hogar, incluso para ver si hay diferencias entre el gasto de los hogares con jefes hombres o mujeres o dependiendo del nivel educativo, estimamos la siguiente regresión lineal.

```
modelo ← lm(ln_alimentos ~ ln_ingreso + tot_integ + edad_jefe + sexo_jefe +
clase_hog + educa_jefe, data=

summary(modelo, digits=5)
```

```
##
## Call:
## lm(formula = ln_alimentos ~ ln_ingreso + tot_integ + edad_jefe +
##    sexo_jefe + clase_hog + educa_jefe, data = enigh)
##
## Residuals:
##    Min       1Q       Median    3Q       Max
## -5.2978    -0.3168   0.0594    0.3931   2.9475
##
## Coefficients:
##                            Estimate Std. Error  t value   Pr(>|t|)
## (Intercept)                4.4849033   0.0339524 132.094 < 2e-16 ***
```

```
## ln_ingreso                       0.3933311    0.0034379 114.412 < 2e-16 ***
## tot_integ                        0.0548082    0.0016565 33.087  < 2e-16 ***
## edad_jefe                        -0.0021246   0.0001684 -12.619 < 2e-16 ***
## sexo_jefeMujer                   -0.0587602   0.0051457 -11.419 < 2e-16 ***
## clase_hogNuclear                 0.2042114    0.0081550 25.041  < 2e-16 ***
## clase_hogAmpliado                0.2090755    0.0103082 20.282  < 2e-16 ***
## clase_hogCompuesto               0.2277305    0.0290524 7.839   4.62e-15 ***
## clase_hogCorresidente            0.3032837    0.0372289 8.146   3.80e-16 ***
## educa_jefePreescolar             0.0718997    0.0692735 1.038     0.299
## educa_jefePrimaria incompleta    0.0844366    0.0098432 8.578  < 2e-16 ***
## educa_jefePrimaria completa      0.1391208    0.0101146 13.754 < 2e-16 ***
## educa_jefeSecundaria incompleta 0.1574192    0.0150112 10.487 < 2e-16 ***
## educa_jefeSecundaria completa    0.1831877    0.0101396 18.067 < 2e-16 ***
## educa_jefePreparatoria incompleta 0.2536362  0.0154350 16.433 < 2e-16 ***
## educa_jefePreparatoria completa 0.2463678    0.0116344 21.176 < 2e-16 ***
## educa_jefeProfesional incompleta 0.2741757   0.0166326 16.484 < 2e-16 ***
## educa_jefePresional completa     0.3418831    0.0124803 27.394 < 2e-16 ***
## educa_jefePosgrado               0.4346385    0.0201058 21.618 < 2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5995 on 74184 degrees of freedom
## Multiple R-squared: 0.3414, Adjusted R-squared: 0.3413
## F-statistic: 2137 on 18 and 74184 DF, p-value: < 2.2e-16
```

Con estos resultados podemos decir que en *promedio* (para todos los hogares), si un hogar aumenta de tamaño en una persona el gasto en alimentos aumentaría en 5.48% al trimestre. Esto lo sabemos porque

```
tot_integ                    0.0548082
```

Los resultados del modelo de regresión lineal efectuado, también muestran que todas las variables son significativas, excepto *educa_jefePreescolar*. Esta variable contiene los efectos diferenciados sobre el gasto en alimentos que tendría un hogar donde la educación del jefe del hogar sea

preescolar. Para detallar un poco más sobre los coeficientes obtenidos y su significado, considera el caso de la variable sexo. Vemos que la regresión arroja los siguientes resultados;

```
sexo_jefeMujer              -0.0587602 0.0051457 -11.419 < 2e-16 ***
```

Esto quiere decir, que el gasto en alimentos de un hogar con jefa de familia mujer es 5.87% menos comparado con los hogares que tienen jefe de familia hombre. Recuerda que esta comparación se hace considerando el valor de la categoría de referencia. En este caso hombres. Además, entre mas años tenga el jefe del hogar, menor es el gasto en alimentos.

Antes de concluir debemos revisar el ajuste del modelo tal como lo vimos con anterioridad, debemos analizar los residuos estimados.

```
predict<- enigh %>%
     add_predictions(modelo) %>%
     add_residuals(modelo)
```

Los residuos son de media cero, aunque un poco sesgados, aunque dado que tenemos una muestra grande esto no representa un problema. Tampoco existe una correlación entre los residuos y las variables independientes.

```
predict %>%
     summarise(
          media_ui = mean(resid,na.rm=TRUE),
          asimetria_ui = skewness(resid,na.rm=TRUE),
          kurtosis_ui = kurtosis(resid,na.rm=TRUE),
          cov_x1_ui = cov(ln_ingreso, resid),
          cov_x2_ui = cov(tot_integ, resid)
          )
## # A tibble: 1 x 5
##    media_ui   asimetria_ui     kurtosis_ui cov_x1_ui  cov_x2_ui
##    <dbl>      <dbl>            <dbl>       <dbl>      <dbl>
## 1  2.94e-13   -0.962           6.04        -1.20e-15  2.49e-15
```

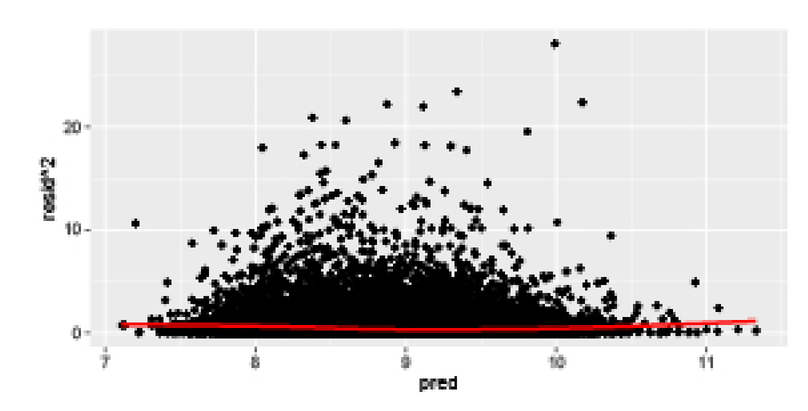
```
hist(predict$resid)
```

Histogram of predict$resid

La gráfica de los residuos contra la variable dependiente estimada, tampoco muestra signos de algún problema.

```
predict %>%
     ggplot(aes(x = pred, y = resid^2)) +
     geom_point()+
geom_point() +
geom_smooth(color = "red", se = FALSE)
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



4 Ejemplo 2: Visitantes extranjeros por entrada aérea

La Secretaría de Turismo en México publica de manera mensual los visitantes extranjeros que llegan al país por vía aérea. Estos datos se encuentran en el archivo visitantes.csv. Analicemos ahora la llegada de visitantes y el efecto de los meses.

Al cargar los datos y ver una descripción general de las variables, observamos que los datos son mensuales desde 2012 a 2020. Además se indica el aeropuerto por el que llego al país, su nacionalidad y sexo.

```
rm(list=ls())
entradas ← read_csv("visitantes.csv")
summary(entradas)
```

```
##        anio           mes              aeropuerto            nacionalidad
##    Min.   :2012    Length:305508    Length:305508        Length:305508
##    1st Qu.:2014    Class :character Class :character     Class :character
##    Median :2016    Mode  :character Mode  :character     Mode  :character
##    Mean   :2016
##    3rd Qu.:2018
##    Max.   :2020
```



```
##        region              sexo             visitantes
##    Length:305508       Length:305508       Min.   :     1.0
##    Class :character    Class :character    1st Qu.:     1.0
##    Mode  :character    Mode  :character    Median :     3.0
##                                            Mean   :   422.4
##                                            3rd Qu.:    17.0
##                                            Max.   :243103.0
```

Después de una rápida revisión de los datos, antes de continuar con un análisis detallado, generemos una nueva tabla de datos que contenga solo el número de visitantes por fecha y filtremos los datos de 2020 ya que debido a la pandemia de la enfermedad covid-19, el turismo, así como las industrias relacionadas, se vieron fuertemente afectadas.

Para generar esta nueva tabla de datos es necesario utilizar algunas herramientas que se aprendieron en otros capítulos:

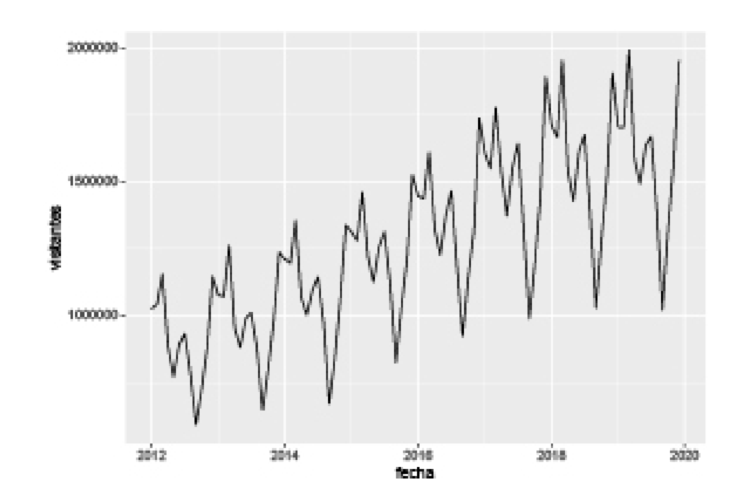
```
# Filtramos los datos del año 2020
entradas ← filter(entradas, anio<2020)
# Generamos una sola tabla con los visitantes por año y mes
entradas ← group_by(entradas, anio,mes)
entradas ← summarise (entradas, visitantes=sum(visitantes), .groups = "drop")

# Reconvertimos los factores de mes a números
entradas ← entradas %>%
mutate(mes_num =mes)
entradas$mes_num ← gsub("Enero", 1, entradas$mes_num)
entradas$mes_num ← gsub("Febrero", 2, entradas$mes_num)
entradas$mes_num ← gsub("Marzo", 3, entradas$mes_num)
entradas$mes_num ← gsub("Abril", 4, entradas$mes_num)
entradas$mes_num ← gsub("Mayo", 5, entradas$mes_num)
entradas$mes_num ← gsub("Junio", 6, entradas$mes_num)
entradas$mes_num ← gsub("Julio", 7, entradas$mes_num)
entradas$mes_num ← gsub("Agosto", 8, entradas$mes_num)
entradas$mes_num ← gsub("Septiembre", 9, entradas$mes_num)
entradas$mes_num ← gsub("Octubre", 10, entradas$mes_num)
entradas$mes_num ← gsub("Noviembre", 11, entradas$mes_num)
entradas$mes_num ← gsub("Diciembre", 12, entradas$mes_num)
entradas ← entradas %>%
     mutate(mes_num=as.numeric(mes_num))

# Generamos una sola variable de fecha
entradas ← mutate(entradas, fecha=make_date(anio, mes_num, 1))
```

Graficamos el número de visitantes para ver el comportamiento de la información. Se observa claramente un patrón estacional, analicemos más a detalle este hecho. ¿Cuáles meses son más atractivos para que los extranjeros visiten México?, podemos suponen por adelantado que es en los periodos vacacionales, pero en que medida afectan a los visitantes, para tener una métrica sobre ello podemos estimar un modelo de regresión lineal.

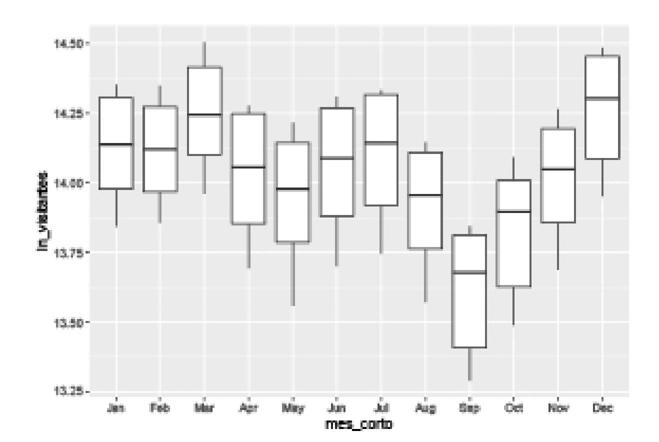
```r
ggplot(entradas, aes(fecha, visitantes)) +
geom_line()
```



Generamos una variable de mes, a través de la fecha. Además analicemos el logaritmo de los visitantes en vez del total de visitantes. Dado que es una variable de personas, no tenemos problema en aplicar logaritmo natural. Con el siguiente gráfico de caja se observa claramente el patrón estacional de los viajeros. Se observa que el mes de septiembre es donde se tienen menos visitantes, mientras que durante los meses de abril y diciembre aumentan.

```r
entradas <- entradas %>%
     mutate(anio2=as.factor(anio)) %>%
     mutate(mes_corto=month(fecha, label=TRUE)) %>%
     mutate(ln_visitantes=log(visitantes))

ggplot(entradas, aes(mes_corto, ln_visitantes)) +
     geom_boxplot()
```



Estimemos la siguiente regresión para ver en que porcentaje varían los visitantes según el mes de referencia.

```r
modelo <- lm(ln_visitantes ~ mes, data = entradas)
summary(modelo)
```

```
##
## Call:
## lm(formula = ln_visitantes ~ mes, data = entradas)
##
## Residuals:
##    Min        1Q        Median     3Q        Max
## -0.38405     -0.19953   0.02714    0.21360   0.27305
##
## Coefficients:
##                Estimate Std.   Error t    value     Pr(>|t|)
## (Intercept)    14.02773        0.07752    180.954   < 2e-16 ***
## mesAgosto      -0.11048        0.10963    -1.008    0.316487
## mesDiciembre   0.23367         0.10963    2.131     0.035973 *
## mesEnero       0.09792         0.10963    0.893     0.374295
## mesFebrero     0.08568         0.10963    0.782     0.436695
## mesJulio       0.07012         0.10963    0.640     0.524181
## mesJunio       0.02870         0.10963    0.262     0.794111
## mesMarzo       0.22177         0.10963    2.023     0.046271 *
## mesMayo        -0.08477        0.10963    -0.773    0.441574
## mesNoviembre   -0.01040        0.10963    -0.095    0.924653
## mesOctubre     -0.19583        0.10963    -1.786    0.077659 .
## mesSeptiembre  -0.41144        0.10963    -3.753    0.000321 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2193 on 84 degrees of freedom
## Multiple R-squared: 0.4116, Adjusted R-squared: 0.3345
## F-statistic: 5.342 on 11 and 84 DF, p-value: 2.332e-06
```

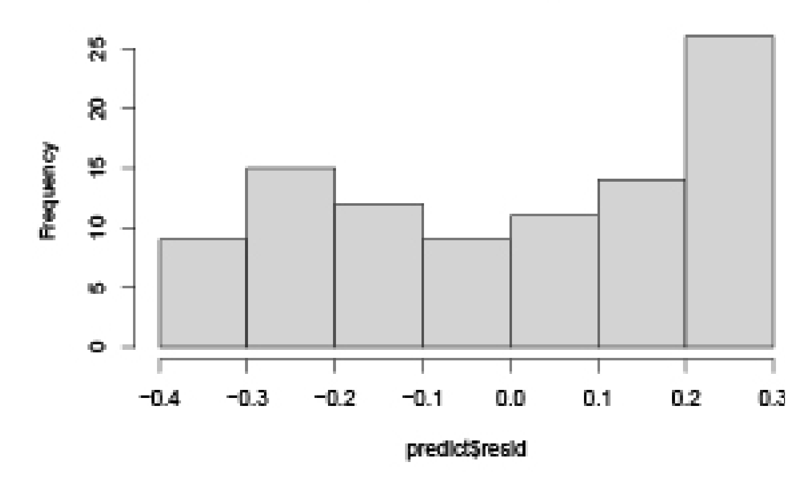
Ahora analicemos los residuos.

```
predict<- entradas %>%
     add_predictions(modelo) %>%
     add_residuals(modelo)

predict %>%
     summarise(
          media_ui = mean(resid),
          asimetria_ui = skewness(resid),
          kurtosis_ui = kurtosis(resid),
)
```

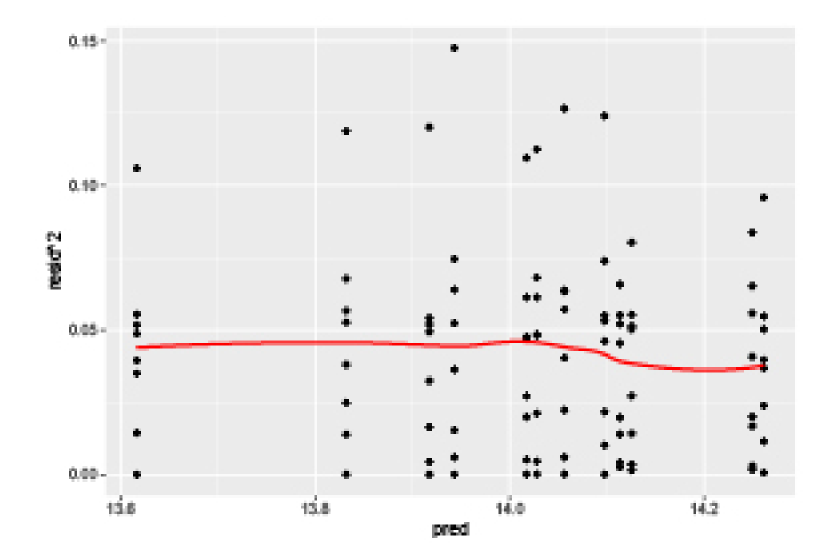
```
## # A tibble: 1 x 3
##   media_ui asimetria_ui kurtosis_ui
##   <dbl>     <dbl>        <dbl>
## 1 -1.89e-15 -0.298       1.64
```

```
hist(predict$resid)
```




```
predict %>%
     ggplot(aes(x = pred, y = resid^2)) +
     geom_point()+
geom_point() +
geom_smooth(color = "red", se = FALSE)
```

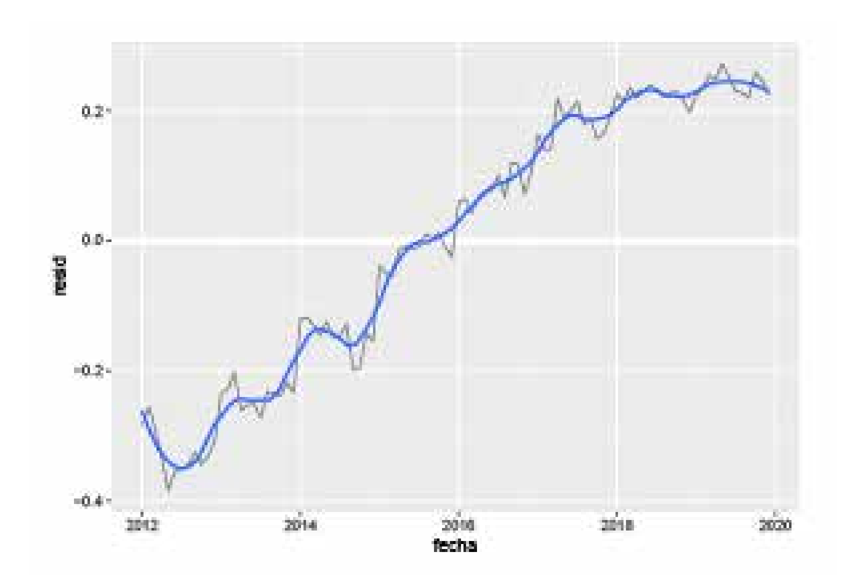
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

Parece ser que existe un patrón creciente en los residuos, claramente, nuestro modelo necesita mejoras. Los residuos muestran una tendencia creciente de largo plazo.

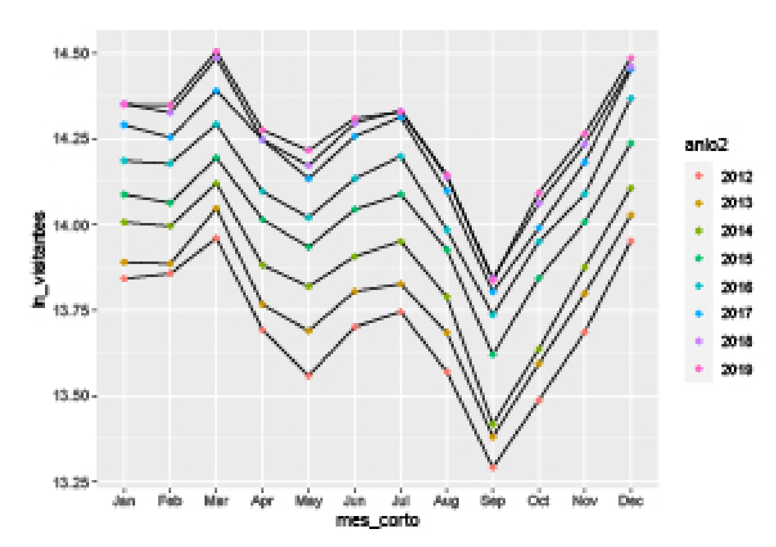
```
predict %>%
ggplot(aes(fecha, resid)) +
geom_ref_line(h = 0) +
geom_line(color = "grey50") +
geom_smooth(se = FALSE, span = 0.20)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Si graficamos los datos por año, podemos observar que cada año el patrón de visitantes es el mismo, baja durante septiembre aumenta en diciembre y marzo. Es probable que este comportamiento de los residuos se deba a que año con año el nivel de los visitantes aumenta, y nuestro modelo falla en capturar este efecto.

```
ggplot(entradas, aes(mes_corto, ln_visitantes))+
geom_line(aes(group = anio2)) +
geom_point(aes(color = anio2))
```



Es posible también a que el efecto estacional de los meses este mal especificado, generemos una variable de estaciones del año para clasificar los meses según el periodo estacional del clima.

```
entradas <- entradas %>%
mutate(estaciones = fct_recode(mes_corto)) %>%
mutate(estaciones = fct_collapse(estaciones,
"primavera" = c("Mar","Apr","May"),
"verano" = c("Jun","Jul","Aug"),
"otoño" = c("Sep","Oct","Nov"),
"invierno" = c("Dec","Jan","Feb")))
```

Es importante una nota en este punto. Si el idioma de R esta en Español, tendrás que escribir;

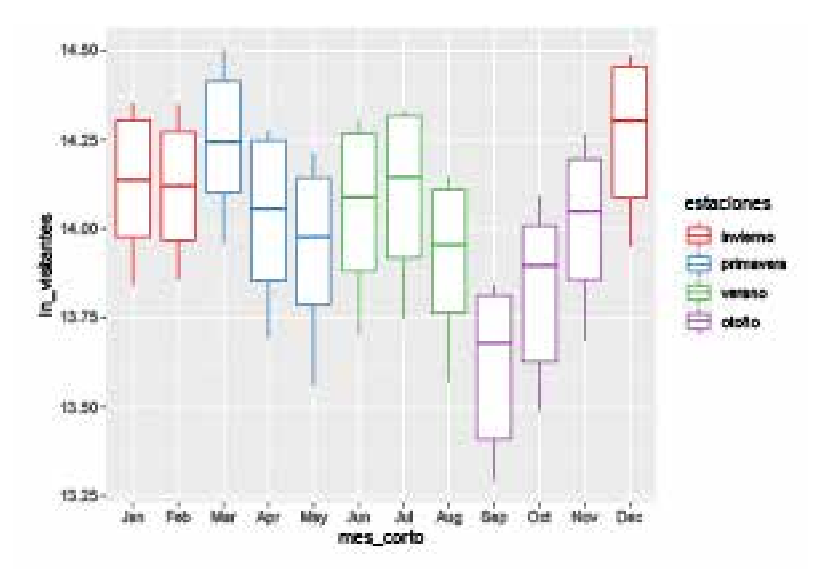
```
entradas <- entradas %>%
mutate(estaciones = fct_recode(mes_corto)) %>%
mutate(estaciones = fct_collapse(estaciones,
"primavera" = c("mar","abr","may"),
"verano" = c("jun","jul","ago"),
"otoño" = c("sep","oct","nov"),
"invierno" = c("dic","ene","feb")))
```

Ahora añadimos a la gráfica de caja identificando a los meses por estaciones. Parece ser que no hay un patrón claro en los visitantes si lo analizamos por estación, pues dentro de una misma estación hay meses donde el flujo de visitantes aumenta y disminuye.

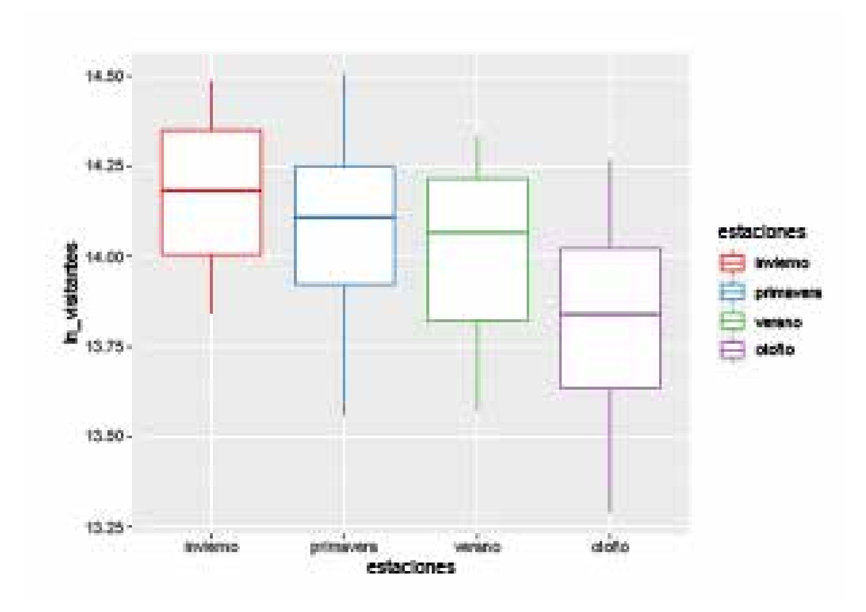
```
entradas %>%
```



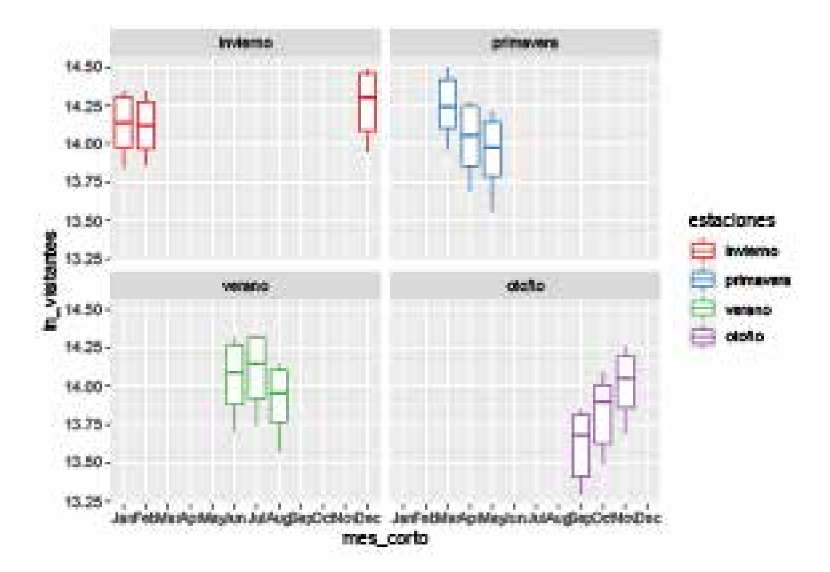
```
ggplot(aes(mes_corto, ln_visitantes, color = estaciones)) +
geom_boxplot() +
     scale_color_brewer(palette = "Set1")
```




```
entradas %>%
ggplot(aes(estaciones, ln_visitantes, color = estaciones)) +
geom_boxplot() +
     scale_color_brewer(palette = "Set1")
```




```
entradas %>%
ggplot(aes(mes_corto, ln_visitantes, color = estaciones)) +
geom_boxplot() +
     scale_color_brewer(palette = "Set1") +
     facet_wrap(~ estaciones)
```



Observa que en la gráfica hemos incluido la opción `scale_color_brewer(palette = "Set1")` esto permite cambiar el color de los puntos. Los detalles de este tipo de modificaciones los aprenderás en el capítulo 17.

Tratemos incluyendo en la regresión el año como variables binarias y probemos nuevamente.

```
modelo2 <- lm(ln_visitantes ~ mes + anio2, data = entradas)
summary(modelo2)
```

```
##
## Call:
## lm(formula = ln_visitantes ~ mes + anio2, data = entradas)
##
## Residuals:
##        Min       1Q   Median        3Q     Max
##  -0.057883 -0.014555 0.001489 0.013962 0.069711
##
## Coefficients:
##                 Estimate Std. Error  t value  Pr(>|t|)
## (Intercept)     13.70156   0.01131   1211.503 < 2e-16 ***
## mesAgosto       -0.11048   0.01271   -8.692   4.72e-13 ***
## mesDiciembre    0.23367    0.01271   18.384   < 2e-16 ***
## mesEnero        0.09792    0.01271   7.704    3.77e-11 ***
## mesFebrero      0.08568    0.01271   6.741    2.57e-09 ***
## mesJulio        0.07012    0.01271   5.516    4.46e-07 ***
## mesJunio        0.02870    0.01271   2.258    0.0268 *
```



```
## mesMarzo        0.22177    0.01271   17.447   < 2e-16 ***
## mesMayo         -0.08477   0.01271   -6.669   3.50e-09 ***
## mesNoviembre    -0.01040   0.01271   -0.818   0.4158
## mesOctubre      -0.19583   0.01271   -15.407  < 2e-16 ***
## mesSeptiembre   -0.41144   0.01271   -32.369  < 2e-16 ***
## anio22013       0.08771    0.01038   8.451    1.37e-12 ***
## anio22014       0.18025    0.01038   17.368   < 2e-16 ***
## anio22015       0.30988    0.01038   29.859   < 2e-16 ***
## anio22016       0.40740    0.01038   39.255   < 2e-16 ***
## anio22017       0.50578    0.01038   48.734   < 2e-16 ***
## anio22018       0.55060    0.01038   53.053   < 2e-16 ***
## anio22019       0.56773    0.01038   54.703   < 2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02542 on 77 degrees of freedom
## Multiple R-squared: 0.9927, Adjusted R-squared: 0.9911
## F-statistic: 585.7 on 18 and 77 DF, p-value: < 2.2e-16
```

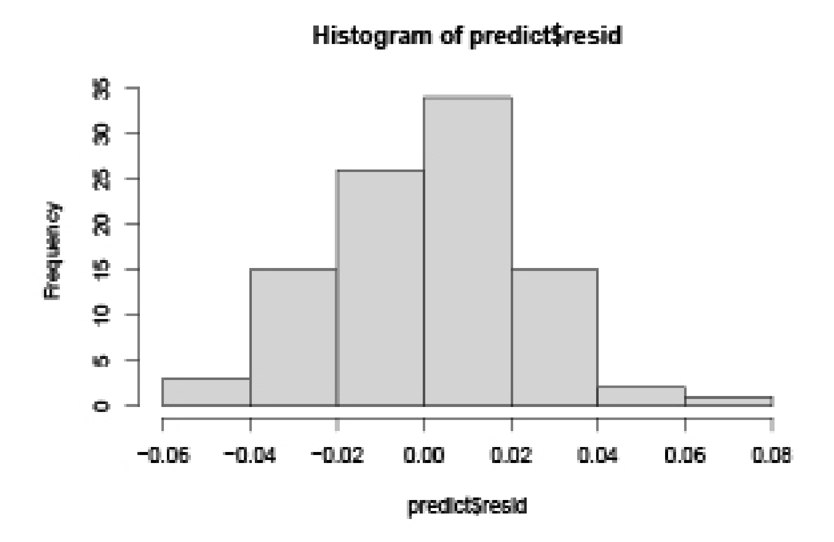
Generemos los residuos del modelo y observamos que los residuos tienen forma acampanada y que su relación con la variable de predicción es casi cero. Estos son buenos elementos para decidir sobre la eficacia del modelo.

```
predict<- entradas %>%
     add_predictions(modelo2) %>%
     add_residuals(modelo2)

predict %>%
  summarise(
     media_ui = mean(resid,na.rm=TRUE),
     asimetria_ui = skewness(resid,na.rm=TRUE),
     kurtosis_ui = kurtosis(resid,na.rm=TRUE),
     )
```

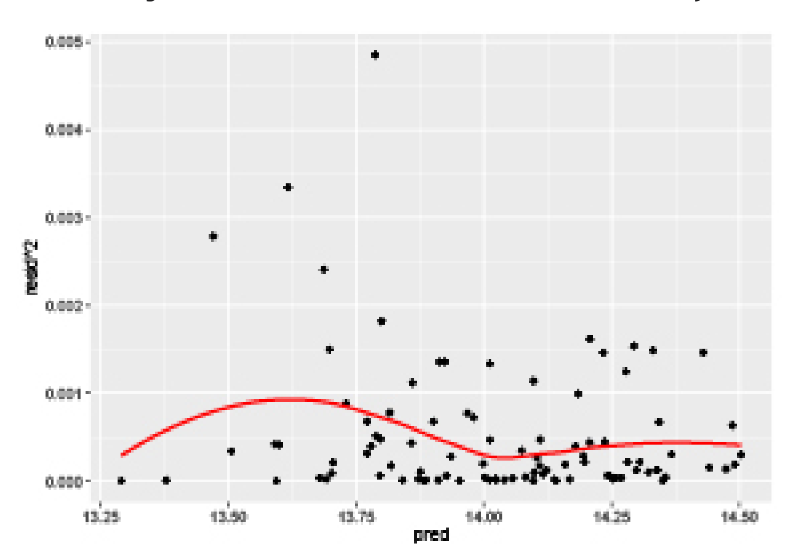
```
## # A tibble: 1 x 3
## media_ui asimetria_ui kurtosis_ui
## <dbl> <dbl> <dbl>
## 1 1.11e-16 0.00996 3.25
```

```
hist(predict$resid)
```




```
predict %>%
     ggplot(aes(x = pred, y = resid^2)) +
     geom_point()+
geom_point() +
geom_smooth(color = "red", se = FALSE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Note que la variable de mes base es abril, por lo tanto, la interpretación de las variables de mes, se deben interpretar comparadas con esta referencia. Podemos concluir que durante septiembre, en promedio, el flujo de visitantes es 41% menor que cuando es abril. Esto lo sabemos porque

```
mesSeptiembre -0.41144      0.01271 -32.369  < 2e-16 ***
```


En el caso del mes de noviembre, tenemos

```
mesNoviembre -0.01040       0.01271 -0.818   0.4158
```

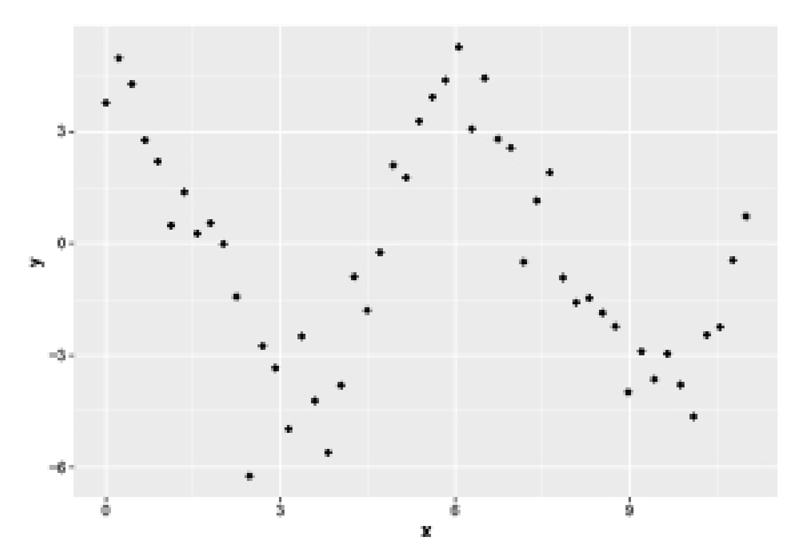
Esto nos indica que la variable no es significativa. Es decir, en promedio no hay diferencias entre los visitantes en los meses de Abril y de Noviembre

6 Modelos de orden superior

Hemos hablado de los modelos lineales los cuales constituyen la base de los modelos de análisis y son los mas usados. **R** nos permite ajustar otros tipos de modelos a un conjunto de datos. Por ejemplo, considera que tiene un conjunto de datos con este comportamiento

```
simulado <- tibble(
x = seq(0, 3.5 * pi, length = 50),
y = 4 * cos(x) + rnorm(length(x))
)

ggplot(simulado , aes(x, y)) +
geom_point()
```



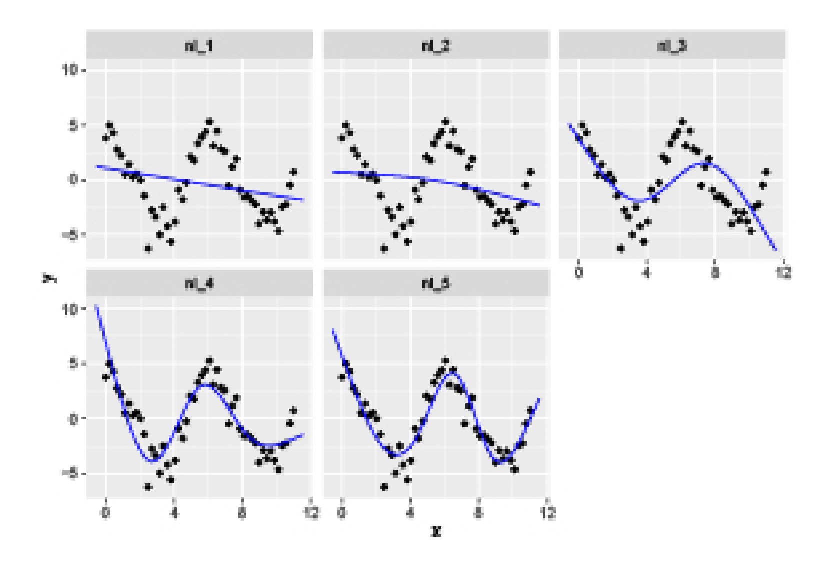
Al ver este conjunto de datos sabemos que no es posible un ajuste en forma de lineal entre *x,y*. Podemos ajustar modelos de orden superior con una pequeña modificación a la función **lm()**

```
nl_1 <- lm(y ~ ns(x, 1), data = simulado)
nl_2 <- lm(y ~ ns(x, 2), data = simulado)
nl_3 <- lm(y ~ ns(x, 3), data = simulado)
nl_4 <- lm(y ~ ns(x, 4), data = simulado)
nl_5 <- lm(y ~ ns(x, 5), data = simulado)
```

Hemos generado 5 modelos de orden superior para el conjunto de datos. Para cada uno de ellos podemos agregar las predicciones usando el comando **gather_predictions**

```
todos ← simulado %>%
data_grid(x = seq_range(x, n = 50, expand = 0.1)) %>%
gather_predictions(nl_1, nl_2, nl_3, nl_4, nl_5, .pred = "y")
```

```
ggplot(simulado, aes(x, y)) +
     geom_point() +
     geom_line(data = todos, color = "blue")+
     facet_wrap(~ model)
```



Decidir el mejor modelo para un conjunto de datos es una tarea de la estadística y casi siempre se relacionará con el comportamiento de los errores que produce el modelo. De la gráfica podemos ver como el modelo 1, de seguro tendrá errores muy amplios, en comparación con el modelo 5. Podemos calcular estos errores usando **gather_residuals**

```
todos ← simulado %>%
gather_residuals(nl_1, nl_2, nl_3, nl_4, nl_5, .resid = "resid")
```

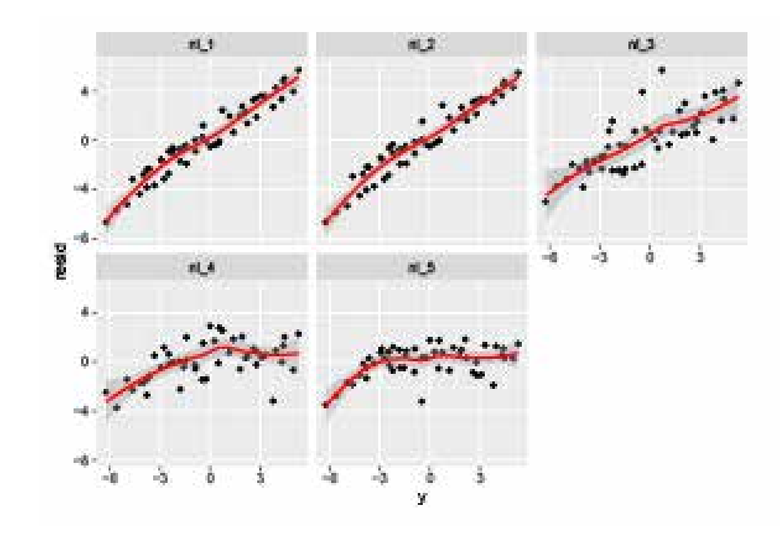
Al graficar la relación entre el valor que predice el modelo y los residuales, confirmamos que el modelo 5, es el mas cercano a mostrar una relación cero entre ellos.

```
ggplot(todos, aes(y, resid))+
     geom_point() +
```



```
          facet_wrap(~ model)+
     geom_smooth(color="red")
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



7 Comentarios finales

Existen otros métodos de estimar y corregir algunos problemas que se puedan encontrar con los residuos. Por ejemplo, ajustar las varianzas cuando no son constantes, como el método de White. Además, dependiendo del problema, se debe elegir la herramienta, si nuestra variable dependiente es binaria o de conteo, es incorrecto estimar un modelo de regresión lineal. Sobre el ejemplo dos, tal vez lo más apropiado sera utilizar un análisis de series de tiempo e incluir las variables rezadas de la variable dependiente en vez de solo variables dicotomicas, y que por cierto, el análisis de series de tiempo tiene diferentes supuestos que el de regresión lineal. En este capítulo no adentramos mucho en estos detalles técnicos, porque nos desvarían del tema y el objetivo es solo mostrar como usar estas herramientas.

8 Actividades

1. Retomando la base de datos de la encuesta *ENIGH*, realiza el mismo análisis que trabajamos en el capítulo cambiando el gasto en alimentos por gasto en salud:

- Identifica que variables están relacionadas con el gasto en salud en los hogares
- Efectúa una gráfica que visualmente permita contestar la siguiente pregunta ¿Existen diferencias en el gasto en salud según el sexo del jefe del hogar?
- Calcula el gasto en salud per cápita (del hogar), diferenciando por tipo de hogar. Expresa el resultado como una gráfica
- Propón un modelo de regresión lineal que explique el gasto en salud y revisa su ajuste
- ¿Que conclusiones puedes obtener de este modelo?



Capítulo 15

Modelos con purrr y broom

Capítulo 15 | Modelos con purrr y broom

1 Introducción

En el capítulo anterior mostramos que además de tener que estimar un modelo tenemos que revisar su ajuste, y si es necesario volver a estimarlo cambiando o agregando variables y de nuevo volver a revisar su ajuste, esto puede ser una tarea un tanto abrumadora, sobre todo si tenemos muchos datos con muchos grupos a los cuales se les necesita ajustar un modelo. Afortunadamente existe una forma de facilitar este trabajando haciendo uso de algunas herramientas que aprendimos en otros capítulos.

El objetivo de este apartado es mostrar como usar esas herramientas para facilitar la estimación y análisis de su ajuste, cuando se requiere estimar un modelo para un grupo grande de individuos.

En este capitulo haremos uso de los de esperanza de vida para un grupo de países que se encuentran en el archivo *indicadores_banco_mundial.csv*. Recordemos comenzar nuestro *script* de la manera usual.

En este capitulo utilizaremos las siguientes librerías: modelr, purrr, tidyverse, broom, readr. Todas ellas ya las hemos usado con anterioridad.

```
setwd("~/Dropbox/Curso de R/Cap15_Varios_Modelos")
library (tidyverse)
library (modelr)
library (readr)
library (purrr)
library(broom)
data ← read_csv("indicadores_banco_mudial.csv")
colnames(data)
```



```
## [1] "país Name"
## [2] "país Code"
## [3] "tiempo"
## [4] "tiempo Code"
## [5] "Esperanza de vida al nacer, total (años) [SP.DYN.LE00.IN]"
## [6] "PIB per cápita (US$ a precios constantes de 2010) [NY.GDP.PCAP.KD]"
```

Para facilitar nuestro análisis, realicemos los siguientes cambios en los nombres de las variables

```
names(data)[2] ← "pais"
names(data)[5] ← "esperanza"
names(data)[6] ← "pib_pc"

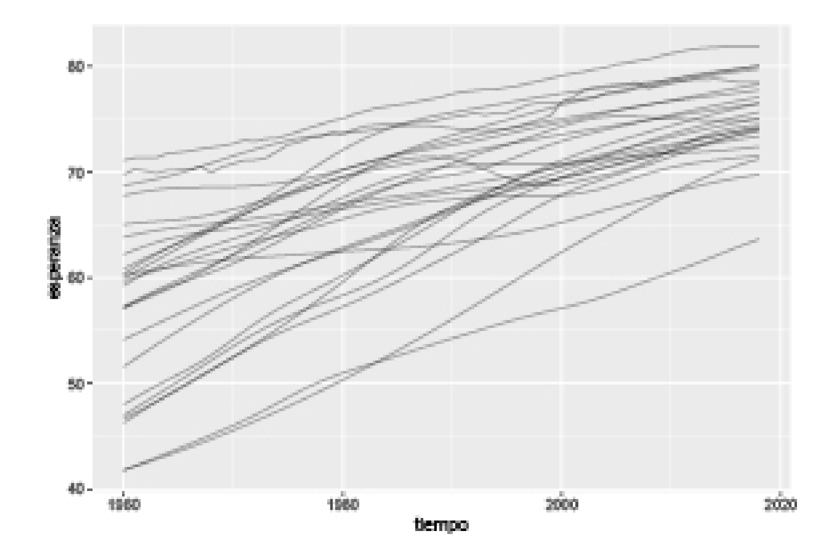
summary(data)
```

```
##    país Name          pais                tiempo         tiempo Code
##    Length:1416        Length:1416         Min.   :1960   Length:1416
##    Class :character   Class :character    1st Qu.:1974   Class :character
##    Mode  :character   Mode  :character    Median :1989   Mode  :character
##                                           Mean   :1989
##                                           3rd Qu.:2004
##                                           Max.   :2018
##    esperanza          pib_pc
##    Min.   :41.76      Min.   : 1005
##    1st Qu.:62.97      1st Qu.: 2396
##    Median :69.08      Median : 4552
##    Mean   :67.52      Mean   : 7809
##    3rd Qu.:73.33      3rd Qu.: 7963
##    Max.   :81.95      Max.   :54833
```

Imagina que deseamos determinar que determina la esperanza de vida. En este caso, nuestra variable dependiente será la esperanza de vida, medida en años. La variable de tiempo indica que tenemos datos desde 1960 hasta 2018. Los datos fueron obtenidos del *databank* del Banco Mundial para un grupo de países del continente americano. Primero analicemos como ha evolucionado la esperanza de vida.

En general, la esperanza de vida a aumentado en todos los países, pero algunos muestran algunas fluctuaciones.

```
data %>%
ggplot(aes(tiempo, esperanza, group = pais)) +
geom_line(alpha = 1/3)
```



La variable pais *Name* indica el nombre, mientras que la variable pais es un nombre corto. Veamos los países que contiene nuestro conjunto de datos.

```
unique(data$`país Name`)
```

```
## [1] "Argentina"                   "Chile"
## [3] "Colombia"                    "México"
## [5] "Belice"                      "Bolivia"
## [7] "Brasil"                      "Canadá"
## [9] "Costa Rica"                  "Estados Unidos"
## [11] "Guatemala"                  "Guyana"
## [13] "Haití"                      "Honduras"
## [15] "Nicaragua"                  "Panamá"
## [17] "Paraguay"                   "Perú"
## [19] "Puerto Rico"                "República Dominicana"
## [21] "San Vicente y las Granadinas" "Suriname"
## [23] "Trinidad y Tobago"          "Uruguay"
```

```
unique(data$pais)
```

```
## [1] "ARG" "CHL" "COL" "MEX" "BLZ" "BOL" "BRA" "CAN" "CRI" "USA" "GTM" "GUY"
```



```
## [13] "HTI" "HND" "NIC" "PAN" "PRY" "PER" "PRI" "DOM" "VCT" "SUR" "TTO" "URY"
```

Supongamos que nos interesa ajustar un modelo lineal de la esperanza de vida para cada país. Realicemos el ejercicio para México ajustando el siguiente modelo.

Primero filtremos el conjunto de datos, luego veamos como evoluciona la esperanza de vida en relación al tiempo.

```
mx <- data %>%
     filter(pais=="MEX")

g1 <- mx %>%
     ggplot(aes(tiempo, esperanza)) +
     geom_line() +
     ggtitle("Datos")
```

Observa que hemos agregado una opción a la gráfica **ggtitle("Datos")** que permite poner un titulo. Ahora generemos un modelo para el caso de México.

```
mx_mod <- lm(esperanza ~ tiempo, data = mx)
```

Una vez generado el modelo, hagamos la predicción de los valores que genera usando **add_predictions** igual que antes. Grafiquemos la relación que predice el modelo.

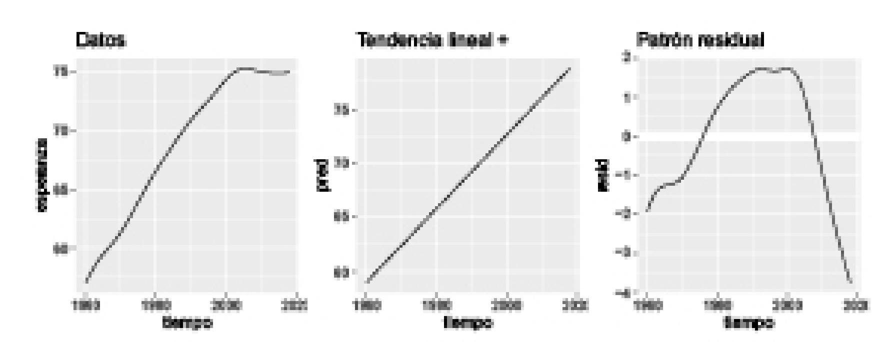
```
g2 <- mx %>%
     add_predictions(mx_mod) %>%
     ggplot(aes(tiempo, pred)) +
     geom_line() +
     ggtitle("Tendencia lineal + ")
```

Finalmente, agreguemos los residuos (diferencias entre lo estimado y el valor real) y grafiquemos.

```
g3 <- mx %>%
     add_residuals(mx_mod) %>%
     ggplot(aes(tiempo, resid)) +
     geom_hline(yintercept = 0, color = "white", size = 3) +
     geom_line() +
     ggtitle("Patrón residual")
```

Las tres gráficas que hemos efectuado se visualizan así

```
library(gridExtra)
grid.arrange(g1, g2, g3, ncol=3)
```



La manera en que hemos presentado estas tres gráficas nos muestra que el conjunto de nuestros datos reales es igual a una tendencia lineal mas un patrón residual o un conjunto de errores, generados por el modelo.

Si queremos un modelo para cada uno de los 24 países, necesitamos facilitar este proceso en lugar de repetir esta tarea 23 veces más.

2 Datos anidados

Para realizar tareas repetitivas es necesario utilizar la función map, de la paquetería purrr. Anteriormente vimos algunos ejemplos en acciones repetitivas para algunos grupos.

El problema es que nuestros datos tienen una estructura diferente, no queremos repetir una tarea para diferentes variables de nuestro conjunto de datos, sino que ahora deseamos estimar un modelo de regresión para diferentes subconjuntos (o grupos, países, etc.) de nuestro conjunto de datos. Para poder realizar esta tarea es necesario generar un conjunto de datos anidados también conocido como nested data frame.

Recordemos anteriormente que el capitulo de vectores, una lista es un vector recursivo que puede contener otras listas, la idea de los datos anidados es similar, es decir, tenemos una tabla de datos agrupada que contiene otras tablas de datos.

Para tener una tabla de datos anidada, necesitamos primero agrupar nuestros datos, en nuestro ejemplo, debemos agrupar la tabla por país y después dar la instrucción nest(), esto hará que por cada país, se agrupen el resto de las variables en una tabla dentro de nuestra de datos:

```
grupo_paises ← data %>%
group_by(pais) %>%
nest()

grupo_paises
```



```
## # A tibble: 24 x 2
## # Groups: pais [24]
##    pais      data
##    <chr>     <list>
## 1  ARG       <tibble [59 x 5]>
## 2  CHL       <tibble [59 x 5]>
## 3  COL       <tibble [59 x 5]>
## 4  MEX       <tibble [59 x 5]>
## 5  BLZ       <tibble [59 x 5]>
## 6  BOL       <tibble [59 x 5]>
## 7  BRA       <tibble [59 x 5]>
## 8  CAN       <tibble [59 x 5]>
## 9  CRI       <tibble [59 x 5]>
## 10 USA       <tibble [59 x 5]>
## # ... with 14 more rows
```

Observa que ahora nuestra tabla que llamamos *grupo_paises* ahora solo tiene 24 observaciones, una por país, y dos variables, la primera indica el nombre del país, la segunda variable llamada data es su vez otra tabla de datos. Si exploramos esta nueva tabla de datos observaremos que ahora esta anidada y se observa mas o menos así:

	pais	data
1	ARG	*5 variables*
2	CHL	*5 variables*
3	COL	*5 variables*
4	MEX	*5 variables*

Ahora que nuestros datos tienen esta estructura, podemos utilizar la instrucción map para repetir la estimación para los diferentes países, es decir, con esta estructura ya podemos repetir una instrucción para cada tabla de datos que contiene el data frame *grupo_paises*.

Antes de ello, y para facilitar aún más el trabajo, escribimos una función que estime el modelo de regresión lineal:

```
modelo_pais <- function(df) {
lm(esperanza ~ pib_pc, data = df)
}
```

Ahora, estimamos con la función *map* la función *modelo_pais* que generamos anteriormente y guardaremos estas estimaciones en un objeto que llamaremos modelos:

```
modelos <- map(grupo_paises$data, modelo_pais)
class(modelos)
## [1] "list"
```

```
length(modelos)
## [1] 24
```

Sí exploramos este objeto notaremos que también es una lista anidada de datos, uno resultado por país. Si ejecutamos la opción *View(modelos)* observaremos lo siguiente

Name	Type	Value
modelos	list [24]	List of length 24
[[1]]	list [12] (S3: lm)	List of length 12
coefficients	double [2]	54.89871 0.00205
residuals	double [59]	-1.401 -1.711 -1.327 -0.454 -1.296 -2.256 ...
effects	double [59]	-545.6051 23.0661 -0.9389 -0.0197 -0.9151 -1.9361 ...
rank	integer [1]	2
fitted.values	double [59]	66.5 66.9 66.6 65.8 66.7 67.8 ...
assign	integer [2]	0 1
qr	list [5] (S3: qr)	List of length 5
df.residual	integer [1]	57
xlevels	list [0]	List of length 0
call	language	lm(formula = esperanza ~ pib_pc, data = df)
terms	formula	esperanza ~ pib_pc
model	list [59 x 2] (S3: data.frame)	A data.frame with 59 rows and 2 columns
[[2]]	list [12] (S3: lm)	List of length 12

Observa que hay 24 modelos, cada uno de ellos tiene en su interior 12 elementos, cada elementos corresponde a un elementos generado por por la regresión lineal efectuada.

Puede resultar confuso analizar muchos modelos si están en un objeto aparte, por lo cual consideramos que es mejor si, como lo hicimos anteriormente, juntamos nuestra tabla de datos con los resultados del modelo:


```
grupo_paises <- grupo_paises %>%
mutate(modelo = map(data, modelo_pais))
grupo_paises
## # A tibble: 24 x 3
## # Groups: pais [24]
##    pais      data               modelo
##    <chr>     <list>             <list>
## 1  ARG       <tibble [59 x 5]> <lm>
## 2  CHL       <tibble [59 x 5]> <lm>
## 3  COL       <tibble [59 x 5]> <lm>
## 4  MEX       <tibble [59 x 5]> <lm>
## 5  BLZ       <tibble [59 x 5]> <lm>
## 6  BOL       <tibble [59 x 5]> <lm>
## 7  BRA       <tibble [59 x 5]> <lm>
## 8  CAN       <tibble [59 x 5]> <lm>
## 9  CRI       <tibble [59 x 5]> <lm>
## 10 USA       <tibble [59 x 5]> <lm>
## # ... with 14 more rows
```

Ahora el objeto **grupo_paises** es un objeto sofisticado, que contiene tablas de datos y un modelo de regresión lineal para el ajuste a ese conjunto de datos.

También podemos predecir los valores residuales y tenerlos en nuestra tabla de datos, tal como lo hicimos anteriormente cuando estimamos solo un modelo.

```
grupo_paises <- grupo_paises %>%
    mutate(resids = map2(data, modelo, add_residuals))

grupo_paises
## # A tibble: 24 x 4
## # Groups: pais [24]
##    pais      data                  modelo resids
##    <chr>     <list>                <list> <list>
## 1  ARG       <tibble [59 x 5]>     <lm>  <tibble [59 x 6]>
## 2  CHL       <tibble [59 x 5]>     <lm>  <tibble [59 x 6]>
## 3  COL       <tibble [59 x 5]>     <lm>  <tibble [59 x 6]>
## 4  MEX       <tibble [59 x 5]>     <lm>  <tibble [59 x 6]>
```

```
## 5  BLZ        <tibble [59 x 5]>     <lm> <tibble [59 x 6]>
## 6  BOL        <tibble [59 x 5]>     <lm> <tibble [59 x 6]>
## 7  BRA        <tibble [59 x 5]>     <lm> <tibble [59 x 6]>
## 8  CAN        <tibble [59 x 5]>     <lm> <tibble [59 x 6]>
## 9  CRI        <tibble [59 x 5]>     <lm> <tibble [59 x 6]>
## 10 USA        <tibble [59 x 5]>     <lm> <tibble [59 x 6]>
## # ... with 14 more rows
```

Esto tiene una ventaja, por que nos permite mantener el orden de cuales resultados le corresponden a cada país. Sin embargo, para realizar nuestros análisis gráficos es mejor tener nuestros de forma tidy en vez de anidados. Para desagrupar nuestros datos, usamos la función *unnest()*, indicando la tabla que deaseamos desagrupar, en este caso la tabla *resids*:

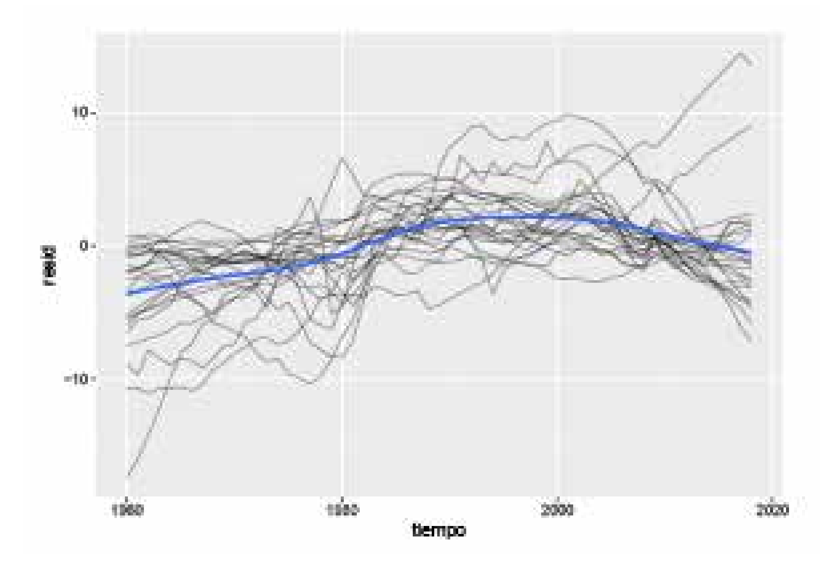
```
residuos <- unnest(grupo_paises, resids)
residuos
```

```
## # A tibble: 1,416 x 9
## # Groups: pais [24]
##    pais data  modelo `país Name` tiempo `tiempo Code` esperanza pib_pc resid
##    <chr> <list> <list> <chr>   <dbl>      <chr>       <dbl>  <dbl>   <dbl>
## 1  ARG  <tibbl~ <lm> Argentina 1960      YR1960      65.1   5643. -1.40
## 2  ARG  <tibbl~ <lm> Argentina 1961      YR1961      65.2   5853. -1.71
## 3  ARG  <tibbl~ <lm> Argentina 1962      YR1962      65.3   5711. -1.33
## 4  ARG  <tibbl~ <lm> Argentina 1963      YR1963      65.3   5323. -0.454
## 5  ARG  <tibbl~ <lm> Argentina 1964      YR1964      65.4   5773. -1.30
## 6  ARG  <tibbl~ <lm> Argentina 1965      YR1965      65.5   6286. -2.26
## 7  ARG  <tibbl~ <lm> Argentina 1966      YR1966      65.6   6152. -1.86
## 8  ARG  <tibbl~ <lm> Argentina 1967      YR1967      65.8   6255. -1.91
## 9  ARG  <tibbl~ <lm> Argentina 1968      YR1968      66.0   6461. -2.14
## 10 ARG  <tibbl~ <lm> Argentina 1969      YR1969      66.2   6981. -2.98
## # ... with 1,406 more rows
```

Nota que el objeto residuos tiene mas de mil filas, porque ahora hemos desagrupado la información. Asi es mas sencillo graficar los datos de los resiudos por pais. Recuerda que estos son un primer indicador del ajuste del modelo. Podemos realizar el gráfico de los residuos:


```
residuos %>%
ggplot(aes(tiempo, resid)) +
geom_line(aes(group = pais), alpha = 1 / 3) +
geom_smooth(se = FALSE)
```

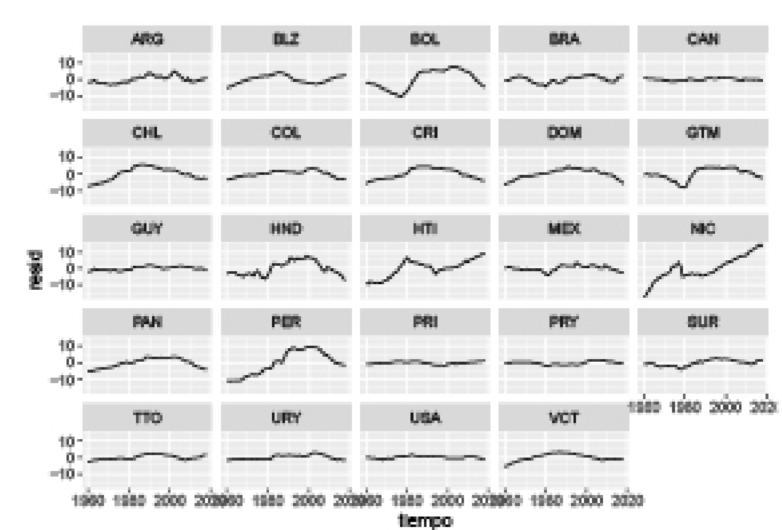
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



En esta gráfica cada linea corresponde a los residuos que genera un modelo lineal para determinado país. La linea azul, es al ajuste suaviado (tendencia considerando todos los modelos)

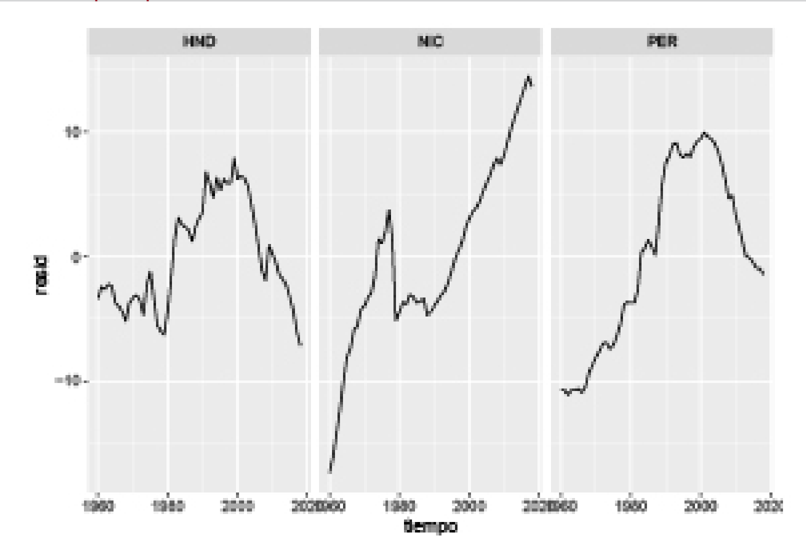
Podemos realizar un gráfico de residuos por país

```
residuos %>%
     ggplot(aes(tiempo, resid)) +
     geom_line() +
     facet_wrap(~pais)
```



Entre mas plana sea la linea significa que es muy probable que hay un mejor ajuste entre las variables que tratamos de explicar. Observa que el país *NIC* tiene una tendencia creciente, y no plana. Lo mismo sucede con *HND* y *PER*. Observemos estos paises por separado.

```
filter(residuos, pais=="NIC" || pais=="HND" || pais=="PER") %>%
     ggplot(aes(tiempo, resid)) +
     geom_line() +
          facet_wrap(~pais)
```



En estos casos, el modelo genera errores mas marcados, indicando que es muy probable que en estos países no exista una relación lineal entre el tiempo y la esperanza de vida.

3 Calidad del modelo con broom

Una forma muy útil para revisar el ajuste de muchos modelos a la vez es a través de la función *glance()*, la cual extrae métricas de ajuste de un modelo y los acomoda en una sola fila:

```
glance(mx_mod)
```

```
## # A tibble: 1 x 12
## r.squared     adj.r.squared    sigma statistic p.value df logLik AIC  BIC
##    <dbl> <dbl>      <dbl>      <dbl>     <dbl>   <dbl> <dbl> <dbl>   <dbl>
## 1  0.933 0.932      1.58       796.  3.48e-35   1     -110. 225.       231.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

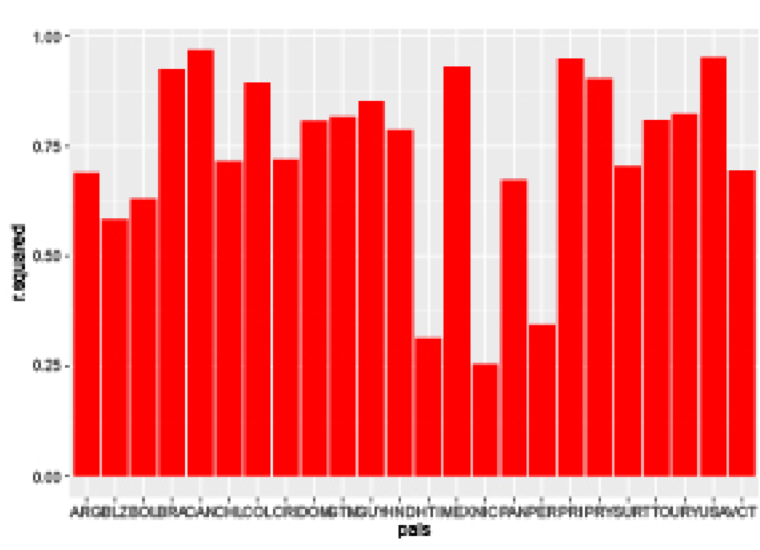
De este conjunto de medidas permiten identificar el ajuste de un modelo. De ellas únicamente trabajaremos igual que antes con del *r.squared*, y por ahora es suficiente entender que entre mayor valor tenga, mejor es el ajuste.

Para emparejar a cada una de estas medidas con el país al que pertenece el modelo que las genera, usaremos esta función en combinación con map y asi podemos visualizar el ajuste de los modelos de forma anidada:

```
grupo_paises <- grupo_paises %>%
mutate(ajustes = map(modelo, glance)) %>%
unnest(ajustes)
```

De esta manera podemos generar generar una gráfica en la que observemos el valor de esta medida de ajuste para cada país.

```
grupo_paises %>%
     ggplot() +
          geom_bar(aes(pais, r.squared), stat = "identity", fill="red")
```



Observa que para los países de NIC, HDN y PER, el indicador es muy bajo, por lo cual los errores queobservamos son muy altos.

Supongamos que deseamos identificar los modelos que tienen una baja *r2*. Reordenamos los modelos según el valor de este indicador:

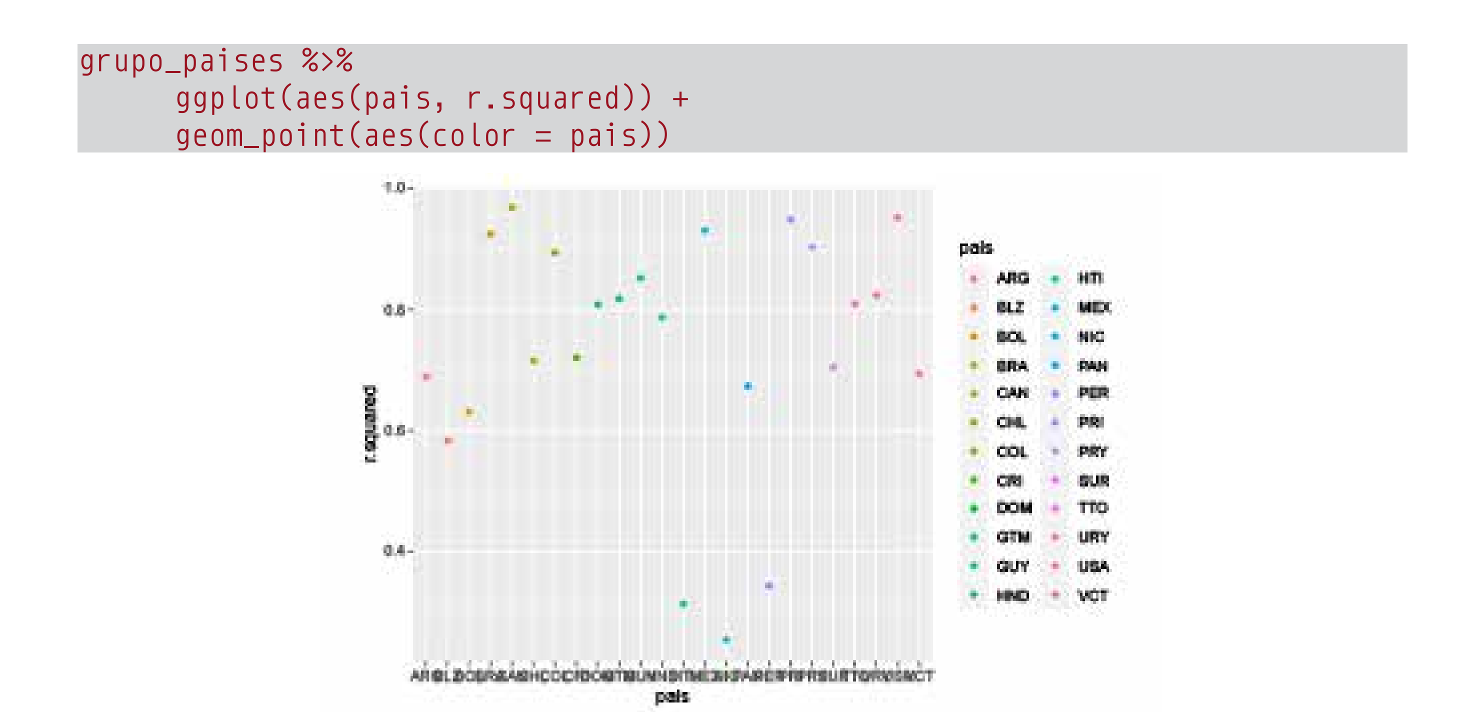
```
grupo_paises %>%
arrange(r.squared)
```

```
## # A tibble: 24 x 16
## # Groups: pais [24]
##    pais data modelo resids r.squared adj.r.squared sigma statistic p.value
##    <chr> <lis> <list> <list> <dbl>        <dbl>      <dbl> <dbl> <dbl>
```

```
## 1  NIC   <tib~ <lm>  <tibb~ 0.254         0.241      7.36  19.4  4.64e- 5
## 2  HTI   <tib~ <lm>  <tibb~ 0.313         0.301      5.24  26.0  4.02e- 6
## 3  PER   <tib~ <lm>  <tibb~ 0.344         0.332      7.23  29.9  1.06e- 6
## 4  BLZ   <tib~ <lm>  <tibb~ 0.583         0.576      2.33  79.7  2.01e-12
## 5  BOL   <tib~ <lm>  <tibb~ 0.631         0.624      5.76  97.3  6.17e-14
## 6  PAN   <tib~ <lm>  <tibb~ 0.673         0.667      2.93  117.  1.87e-15
## 7  ARG   <tib~ <lm>  <tibb~ 0.689         0.683      2.05  126.  4.48e-16
## 8  VCT   <tib~ <lm>  <tibb~ 0.694         0.689      2.16  129.  2.76e-16
## 9  SUR   <tib~ <lm>  <tibb~ 0.704         0.699      1.80  136.  1.04e-16
## 10 CHL   <tib~ <lm>  <tibb~ 0.715         0.710      3.87  143.  3.60e-17
## # ... with 14 more rows, and 7 more variables: df <dbl>, logLik <dbl>,
## #  AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>, nobs <int>
```

Podemos graficarlos para identificar visualmente los países que tienen un ajuste bajo.

```
grupo_paises %>%
     ggplot(aes(pais, r.squared)) +
     geom_point(aes(color = pais))
```



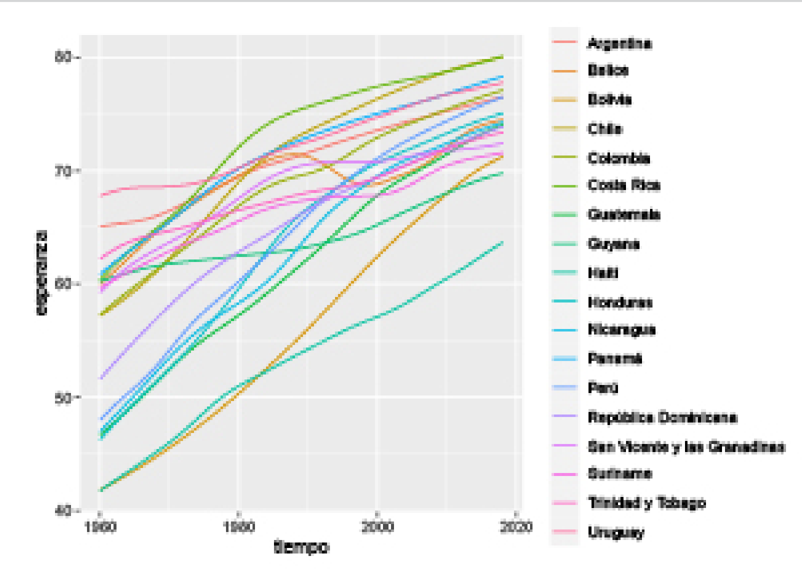
Podemos filtrar los modelos que tengan una r2 menor a 0.9:

```
bajo_ajuste ← filter(grupo_paises, r.squared < 0.9)

data %>%
     semi_join(bajo_ajuste, by = "pais") %>%
```



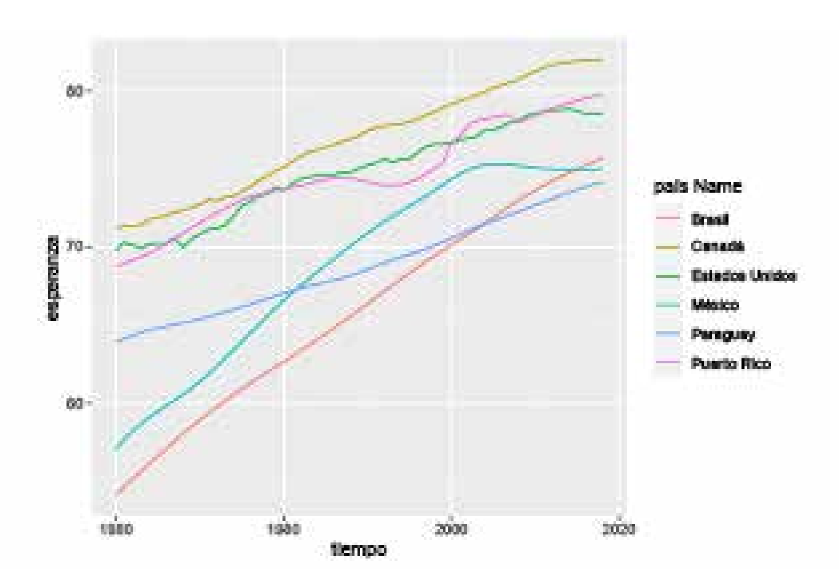
```
     ggplot(aes(tiempo, esperanza, color =`país Name`)) +
     geom_line()
```



O bien filtrar aquellos cuyo ajuste sea mayor a 0.9

```
bajo_ajuste ← filter(grupo_paises, r.squared >0.9)

data %>%
     semi_join(bajo_ajuste, by = "pais") %>%
     ggplot(aes(tiempo, esperanza, color =`país Name`)) +
     geom_line()
```



Estos serian los países con los cuales el modelo tiene el mejor ajuste.

4 Otras utilidades de datos anidados

Existen otras ventajas de utilizar datos anidados, sobre todo si tenemos grandes bases de datos. Por ejemplo, al agrupar los datos por país, podemos agrupar variables y analizar de manera separada sin tener que crear en nuestro ambiente de trbajo muchos objetos.

Si bien, nuestra base de datos de ejemplo no es "grande", podemos usarla de ejemplo, supongamos que nos interesa agrupar las variables por país, no es necesario especificar la función *group_by()*, solo debemos usar la función *nest()* y entre paréntesis indicar las variables que se desean anidar, las variables excluidas serán las que funcionen como individuos.

```
base_pais ← data %>%
nest(tiempo:pib_pc)
```

En ese caso hemos pedido un anidado desde la variable tiempo, hasta la variable pib_pc. Prueba ejecutar lo siguiente, visualiza el resultado y ve las diferentes anidaciones.

```
base_pais ← data %>%
nest(tiempo:esperanza)
```

Es probable que en ambos casos *R* te arroje un warning, sólo para asegurarse lo que quisiste decir es correcto. Esto sucede porque *R* considera que es mejor que nombres **todos** los elementos que quieres anidar.

También podemos utilizar los datos anidados para realizar resúmenes estadísticos por país de manera agrupada.

Supongamos que nos interesa conocer el valor promedio de las variables *esperanza* y *pib_pc*. Podemos utilizar la función *summarise()* de la siguiente manera;

```
resumen ← data %>%
group_by(pais) %>%
     summarise(mean(esperanza), mean(pib_pc), .groups="drop")
head(resumen)
```

```
## # A tibble: 6 x 3
##    pais `mean(esperanza)` `mean(pib_pc)`
##    <chr>      <dbl>           <dbl>
## 1  ARG        71.0            7877.
## 2  BLZ        68.9            2787.
## 3  BOL        56.0            1596.
## 4  BRA        65.8            7881.
```



```
## 5  CAN        76.8            34984.
## 6  CHL        71.1            7651.
```

Pero imaginemos que nos interesa también el valor mínimo, tendríamos que modificar la instrucción anterior como sigue:

```
resumen ← data %>%
group_by(pais) %>%
     summarise(mean(esperanza), mean(pib_pc),min(esperanza), min(pib_pc),
     .groups="drop")
head(resumen)
```

```
## # A tibble: 6 x 5
##    pais `mean(esperanza)` `mean(pib_pc)` `min(esperanza)` `min(pib_pc)`
##    <chr>      <dbl>           <dbl>      <dbl>                <dbl>
## 1  ARG        71.0            7877.      65.1                 5323.
## 2  BLZ        68.9            2787.      60.0                 1079.
## 3  BOL        56.0            1596.      41.8                 1005.
## 4  BRA        65.8            7881.      54.1                 3417.
## 5  CAN        76.8            34984.     71.1                 16406.
## 6  CHL        71.1            7651.      57.2                 3612.
```

¿Qué pasaría si nos interesan aún más estadísticos?. En vez de tener que estar modificando y agrandando la instrucción, podemos hacer una función que calcule los estadísticos que nos interesa, tal como lo hicimos en el capítulo 13. Tomemos de ejemplo la función *summary()*, la cual aplicada a una variable nos proporciona seis medidas de localización.

Intentemos el **summary** de manera conjunta

```
resumen ← data %>%
group_by(pais) %>%
     mutate(Est=summary(esperanza))
```

¿Porqué obtenemos este error?. Esta función da por resultado seis elementos, pero la celda solo admite un elemento. Es decir, para poder calcular y guardar todos los estadísticos que nos interesan necesitamos anidar los resultados.

```
resumen ← data %>%
group_by(pais) %>%
```

```
summarize(Est_Esp_Vida = list(tibble(EV=as.numeric(summary(esperanza)),
               Medida=c("Min","1Q","Mediana","Media","3Q","Max"))),
          Est_Pip_pc = list(tibble(PIB=as.numeric(summary(pib_pc),
               Medida=c("Min","1Q","Mediana","Media","3Q","Max")))),
          .groups="drop")
resumen
```

```
## # A tibble: 24 x 3
##    pais       Est_Esp_Vida         Est_Pip_pc
##    <chr>      <list>               <list>
## 1  ARG        <tibble [6 x 2]>     <tibble [6 x 1]>
## 2  BLZ        <tibble [6 x 2]>     <tibble [6 x 1]>
## 3  BOL        <tibble [6 x 2]>     <tibble [6 x 1]>
## 4  BRA        <tibble [6 x 2]>     <tibble [6 x 1]>
## 5  CAN        <tibble [6 x 2]>     <tibble [6 x 1]>
## 6  CHL        <tibble [6 x 2]>     <tibble [6 x 1]>
## 7  COL        <tibble [6 x 2]>     <tibble [6 x 1]>
## 8  CRI        <tibble [6 x 2]>     <tibble [6 x 1]>
## 9  DOM        <tibble [6 x 2]>     <tibble [6 x 1]>
## 10 GTM        <tibble [6 x 2]>     <tibble [6 x 1]>
## # ... with 14 more rows
```

Observa que en este proceso de anidación primero hemos pedido que los datos que se obtienen del summary sean números y hemos rescrito sus nombres para identificar de que medida se trata.

De esta manera, para cada país tenemos las estadísticas mas importantes de la esperanza de vida y del PIB. Podemos desagruparlas y mostrarlas como dato tipo *tydy*

```
Estadisticas <- resumen %>%
     unnest(c(Est_Esp_Vida, Est_Pip_pc))
Estadisticas
```

```
## # A tibble: 144 x 4
##    pais       EV    Medida     PIB
##    <chr>      <dbl> <chr>      <dbl>
## 1  ARG        65.1  Min        5323.
## 2  ARG        67.8  1Q         7033.
## 3  ARG        71.4  Mediana    7454.
## 4  ARG        71.0  Media      7877.
```



```
## 5  ARG        74.2  3Q         8560.
## 6  ARG        76.5  Max        10883.
## 7  BLZ        60.0  Min        1079.
## 8  BLZ        67.6  1Q         1721.
## 9  BLZ        69.6  Mediana    2607.
## 10 BLZ        68.9  Media      2787.
## # ... with 134 more rows
```

De esta manera podemos manejar con mayor facilidad la estadísticas de nuestros datos.

5 Actividades

1 Realiza un modelo lineal de la esperanza de vida como función del PIB per cápita y calcula lo siguiente:

-El modelo de regresión para cada país. Efectúa un resumen del país 22

-Analiza el ajuste de los residuos de cada país. Di en cuales de ellos es poco probable que exista una relación lineal entre la esperanza de vida y el producto interno bruto

Calcula las métricas de ajuste utilizando *glance*

-Efectúa tres gráfica. En una pon lo modelos que tiene un ajuste menor o igual a .5, en otra los que tienen un ajuste entre .5 y .9. En la tercera incluye aquellos que tienen ajuste mayor a .9 En todos los casos gráfica la relación entre el PIB y la esperanza de vida. Usa la función *ggtitle* para agregar el titulo del gráfico

-Para este conjunto de datos, determina en promedio como ha cambiado la esperanza de vida de 1960 a 2018

Capítulo 16

RMarkdown

Capítulo 16 | RMarkdown

1 Introducción

Hemos visto que con el uso de **#** podemos poner comentarios en el código de **R** y de forma automática se reconocen como texto y no lenguaje de programación. Hemos usado este símbolo en algunos textos que hemos trabajado en este curso. Imagina que estas programando un código y que deseas compartirlo con alguien, de manera que pueda replicar y entender tu trabajo. Tendrás que usar repetidamente este símbolo. O mejor aún, podrías enviarle un documento de word (donde puedes escribir libremente) y copiar y pegar tu código en ese espacio para cuando tu compañero lo reciba copie y peque en código mientras lee el documento en word donde están las explicaciones. Esto suena un poco complicado. **R** tiene determinadas funcionalidades que te permiten compartir no sólo los resultados de tu trabajo, también el procedimiento que usaste para llegar a esos resultados. Todo de una manera fácil y sencilla. **R** también dispone de una herramienta que te permite generar visualizaciones interesantes para la organización de tus datos. En este capítulo trabajaremos estas herramientas. Es importante recalcar que el potencial de **R** en este rubro se encuentra en crecimiento y aquí tocaremos estos profundizando sólo en algunos de los rubros esenciales, pues podríamos desarrollar un curso completo con técnicas para la presentación y visualización de la información.

1.1 Previos

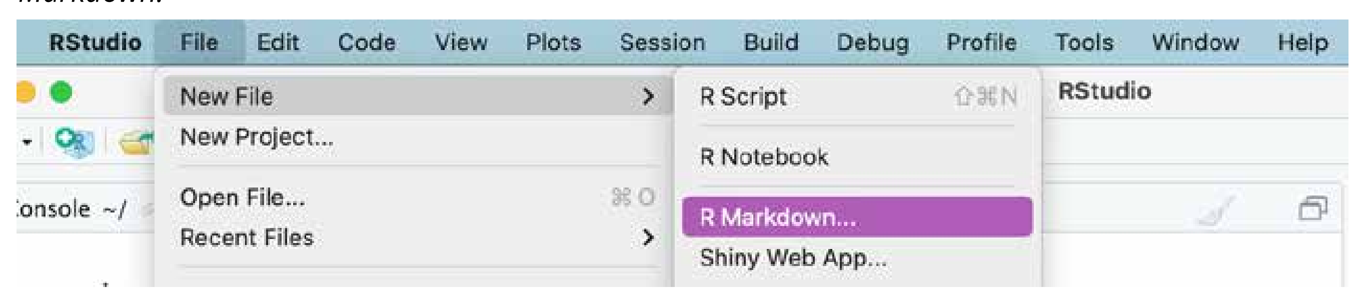
Trabajaremos con diversas paqueterías, la mayoría de ellas se instalarán automáticamente por **R** cuando se necesiten. Puedes adelantar este proceso instalando y activando


```
library(rmarkdown)
library(shiny)
library(DT)
library(flexdashboard)
library(tidyverse)
library(lubridate)
library(babynames)
library(leaflet)
```

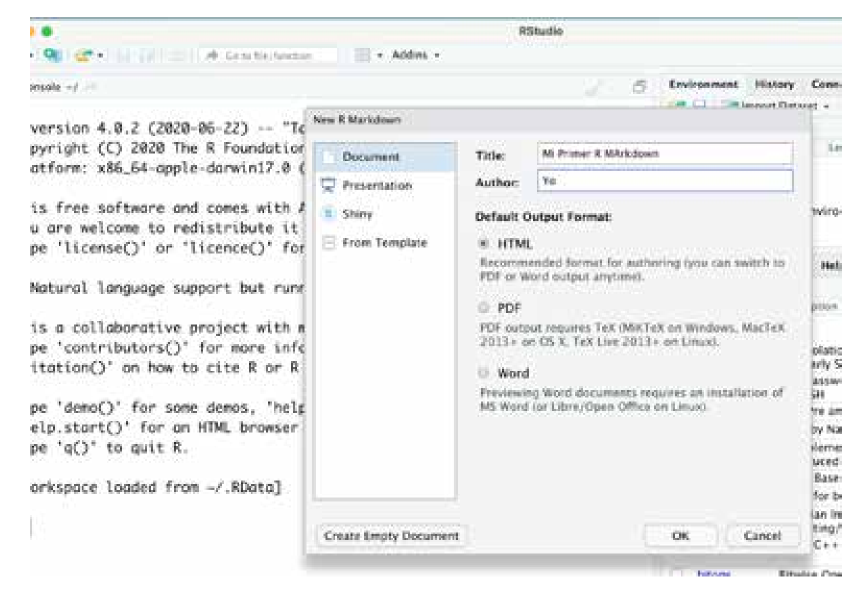
Como siempre es recomendable que guardes las bases de datos de este capítulo en un mismo directorio también guarda ahí todo lo que estaremos trabajando.

2 Básicos del manejo de RMarkdown

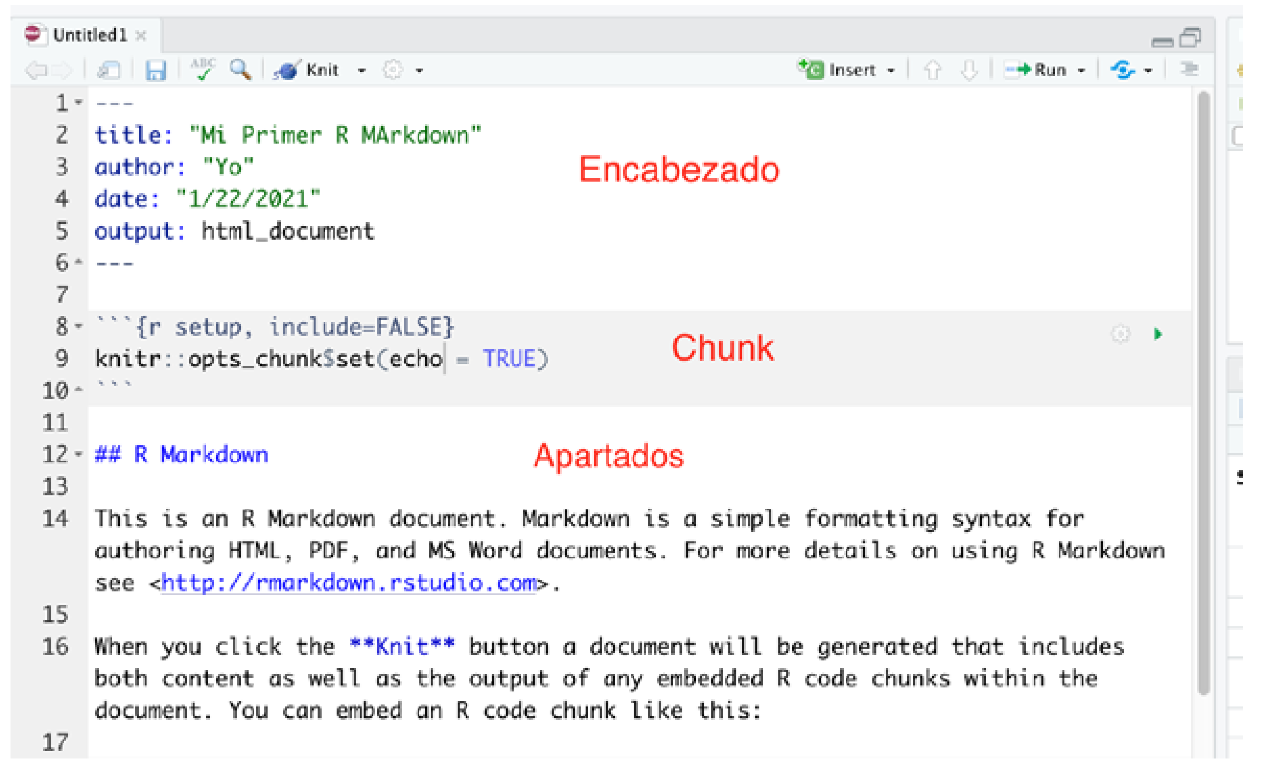
El primer paso para manejar un *RMarkdown** es darlo de alta. Para ello haremos uso del comando *Archivo* o *File*, seleccionaremos nuevo archivo y dentro de las opciones vamos a escoger *R Markdown.*



Lo siguiente que veremos será una pantalla como la siguiente. Donde debemos indicar la información sobre el nombre del documento y el autor, si es que deseamos incluirlo.



Al dar aceptar **R** de forma automática nos muestra un documento con cierta información de **prueba** que se genera de manera automática



Las paqueterías necesarias para este proceso se instalan de forma automática, cuando le indicamos a *R* que vamos a comenzar a trabajar con un documento *RMarkdown*.

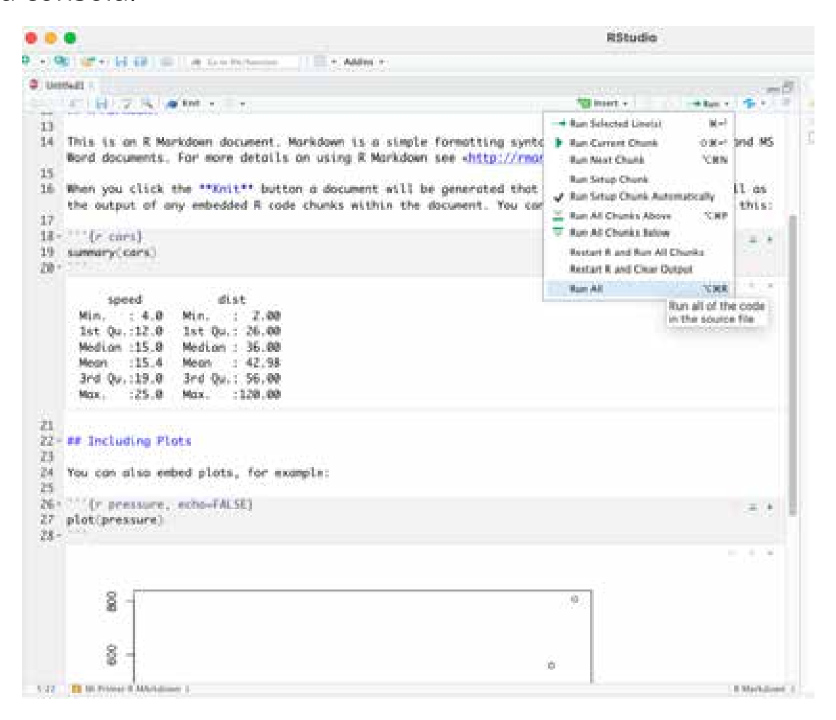
Si este archivo no se genera, seguramente es porque no le fue posible a **R** instalar la paquetería **rmarkdown** o algunas de sus dependencias que son necesarias para trabajar con este tipo de archivos. En este caso **R** te indicará algo mas o menos así *Requiered package versión could not be found*. Si surge este problema en el proceso puedes probar ejecutando el siguiente código;

```
options(repos = c(CRAN="https://cloud.r-project.org"))
install.packages(c("digest", "caTools", "bitops"))
```

Si a pesar de esto el error persiste deberás asegurarte qué la versión de **R** se encuentra actualizada.

El documento generado se compone de tres partes. La primera de ellas es un *Encabezado,* seguida de un *Chunk* y después *Apartados*. Es un documento básico de R tipo *Markdown.* Lo primero que debes saber es que todo lo que sea que tu escribas fuera de un *Chunk* se tratará tal cual, como texto, es decir se interpretará como lenguaje de uso común sin la necesidad de incluir #. Mientras que todo lo que se encuentre dentro de un *Chunk* será interpretado como código. Observa que este documento es muy diferente a los que trabajamos anteriormente denominados *scripts.* Un documento como el recién generado se guarda como una extensión **.Rmd** , mientras que un *script* tiene la extensión **.R**

Si ejecutas la opción *Run All*, observarás que se generan determinadas tablas y una gráfica. Es decir, **R** ha ejecutado el documento y dentro del mismo documento ha mostrado los resultados. Sin necesidad de ir a la consola.



Cuando construyes un documento *.Rmd* en **R** obtienes un cuaderno de trabajo donde código, texto y resultados de la ejecución del código interactúan. Es posible correr cada *Chunk* de manera separada, sólo debes presionar la pequeña flecha verde que se encuentra a la derecha. Cuando ejecutas el *Chunk* **R** efectúa el código y muestra el resultado.

```
```{r cars}
summary(cars)
```

     speed   dist
 Min.   : 4.0   Min.   : 2.00
 1st Qu.:12.0   1st Qu.: 26.00
 Median :15.0   Median : 36.00
 Mean   :15.4   Mean   : 42.98
 3rd Qu.:19.0   3rd Qu.: 56.00
 Max.   :25.0   Max.   :120.00
```

Para indicar a **R** que en un espacio dentro del documento necesitamos un *Chunk* podemos usar la opción insertar *R*

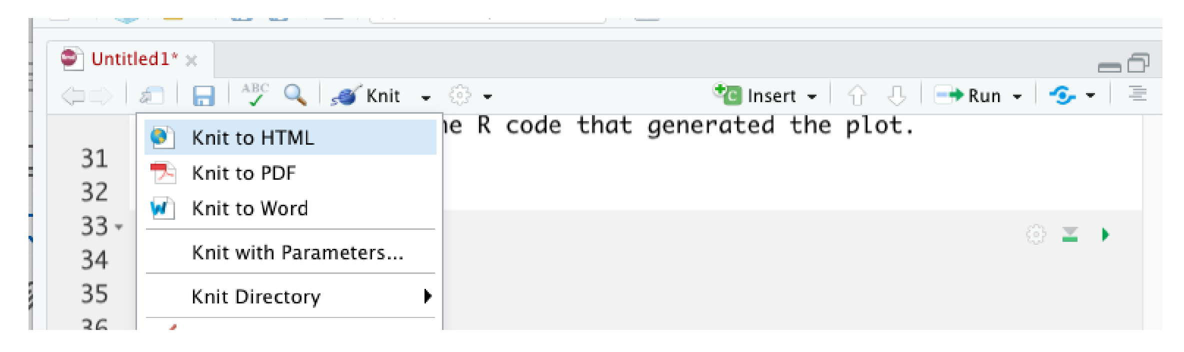


Prueba usando el documento automático generado por **R**, en el mismo genera un *Chunk* y escribe el código mostrado. Luego ejecuta el *Chunk*.

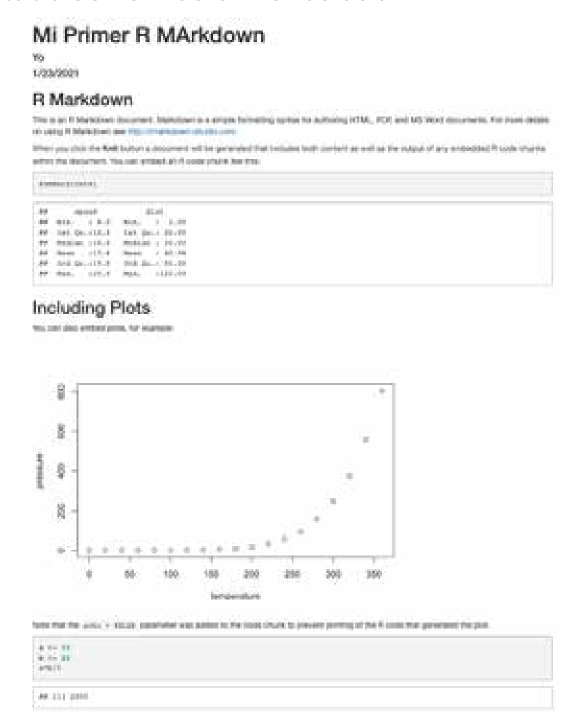
````
```{r}
a <- 50
b <- 80
a*b/2
```

[1] 2000
````

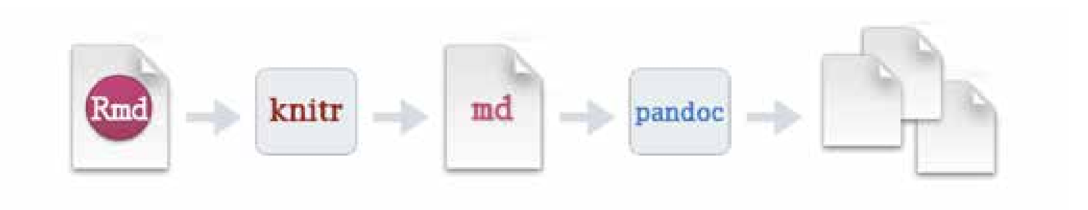
Para generar un reporte completo con todos estos elementos debemos hacer *Knit* to *HTML* al documento.



Esta acción ejecutará un pequeña página HTML con todos los elementos del documento; código, texto y resultados. A ejecutar el *Knit* **R** te pedirá que asignes un nombre y un directorio para guardar el archivo *.Rmd.* El resultado se ve mas o menos así.



Cuando se ejecuta *Knitr* **R** envía el documento *.Rmd* a **KNITR** el cual ejecuta todos los *Chunks* y crea un nuevo documento tipo *md*, que incluye el código, los resultados y el texto. Este proceso se lleva a cabo a través de **PANDOC** que es el encargado de generar el producto final.



La primera parte de este capítulo se enfoca en examinar los tres componentes de *RMarkdown*. En la segunda parte se repasan otras opciones de presentación de resultados alternas a *HTML* y se discuten algunas ideas interactivas de presentación de información.

3 Apartados

Cuando hablamos de apartados nos referimos al texto común que utilizamos en un *Markdown*. Existe algunas características especiales que podemos usar para escribir texto.

3.1 Formatos de texto

Para escribir letras en **Negritas** tenemos dos opciones; ****Negritas** o __Negritas__**

Para escribir letras en *Cursivas* podemos hacer **Cursivas** o *_Cursivas_*

Si deseamos escribir superindices2 podemos hacer **superindicesˆ2ˆ** o **subindices~2~**

3.2 Encabezados

Los encabezados nos permiten incluir nombres de secciones en tres niveles

```
# Nivel 1
## Nivel 2
### Nivel 3
```

Mas adelante veremos como podemos numerar los diferentes niveles de un documento.

3.2.1 Listas

Podemos también generar listas con bulletes

```
* Elemento 1
```

```
* Elemento 2
* Elemento 2a
* Elemento 2ai
* Elemento 2b
```

Esto generará un documento con la indentación a un máximo de tres niveles.

3.3 Ligas e imágenes

Para insertar una imagen debemos escribir ![] seguido del nombre de la imagen entre paréntesis. Esto es **`![](NombreImagen.png)`.** Es importante que la imagen este contenida en el mismo directorio de trabajo que el documento *.Rmd,* en caso contrario debemos incluir el directorio. De manera automática **R** nos muestra la una pequeña visualización de la imagen. Si queremos controlar el tamaño de la imagen podemos incluir **{height=120px}.**



Para insertar una liga debemos indicar entre corchetes [Nombre] el nombre que deseamos tenga la liga y entre () la liga. Es decir [KNITR](https://yihui.org/knitr/) esto nos referencia a la https://yihui.org/knitr/, mientras que en el documento final sólo veremos KNITR en color resaltado.

3.4 Tablas

Podemos efectuar tablas sencillas, escribiéndolas así

```
Columna 1     | Columna 2
--------------|-------------
Contenido 11  | Contenido 12
Contenido 21  | Contenido 22
```

Al ejecutar el documento final, se verán así

Columna 1	Columna 2
Contenido 11	Contenido 12
Contenido 21	Contenido 22

Recuerda que al trabajar con *RMarkdown* una cosa será el documento *.Rmd* sobre el cual trabajamos y otros el documento final que es el resultado de ejecutar el *Knit* con todas las indicaciones que hallamos establecido. Por ahora prueba escribiendo los elementos anteriores en el documento de prueba que genera **R** y después ejecuta *Knitr*

Existen otros elementos que podrás utilizar la trabajar con *RMarkdown* para consultarlos utiliza *Ayuda, Referencia Rápida Markdown*.

4 Chunks

El segundo de los elementos contenidos en un *RMarkdown* son los *Chunk*, en los cuales como ya dijimos, se escribirá todo aquello que deseamos que **R** reconozca cómo código. Una manera fácil y rápida de insertar un *Chunk* en un documento es usar la combinación Cmd/Ctrl-Alt-I. Esto generará el espacio sombreado sobre el cual podemos comenzar a escribir código. Con fines de orden siempre que trabajamos con un *Chunk* es recomendable asignar un nombre. Esto permite desplazarnos en el documento con mayor facilidad.

Para asignar un nombre a determinado *Chunk* sólo es necesario escribirlo en su encabezado. Entenderemos por encabezado del *Chunk*, todo aquello que este contenido dentro de las llaves que acompañan la letra *r*.



Nombrar los *Chunks* nos permite desplazarnos con mayor facilidad en el documento.

4.1 Opciones del *Chunk*

Además del nombre, en el encabezado del *Chunk* es posible incluir algunas otras opciones que nos permiten manipularlo principalmente para decidir sobre su presentación en el documento final.

- **eval=FALSE** cuando incluimos esta opción dentro del encabezado del *Chunk* se entenderá que

deseamos que su contenido no sea evaluado. Esta opción puede ser útil cuando deseamos que se muestre cierto código sin que se incluyan los resultados.

• **include=FALSE** ejecuta el contenido dentro del *Chunk*, pero no lo muestra en el documento final. Tampoco muestra los resultados de su ejecución.

• **echo=FALSE** evita que aparezca el código, pero si muestra los resultados. Esta opción resulta útil cuando sólo nos interesa comunicar el resultado, sin indicar como llegamos a el.

• **message=FALSE** o evita que en el documento final aparezcan mensajes que **R** emite, **warning=FALSE** evita que aparezcan advertencias en el documento.

• **results='hide'** oculta los resultados impreso en el documento final. Si deseamos ocultar las gráficas usamos **fig.sow='hide'**

• **error=TRUE** esta indicación permite que se muestre determinado código en el resultado final, aún y cuando este genere un error.

El diferente material con el que se has trabajado en este curso, ha sido construido usando *RMarkdown*. Cuando construyes un documento con el cual buscas que los demás puedan entender las funcionalidades del código para que puedan por si mismos replicarlas, trabajar con diferentes opciones en el *Chunk* es de gran ayuda.

Por ejemplo, considera el siguiente código

```
df ← tibble::tibble(
a = rlnorm(5),
b = rlnorm(5),
c = rlnorm(5),
d = rlnorm(5)
)
df
```

Si deseamos mostrar sólo el código, pero no el resultado, debemos insertar *Chunk* y en el encabezado el **usamos; {r, eval=FALSE}**

Esto mostrará en el documento final

```
df <- tibble::tibble(
a = rlnorm(5),
b = rlnorm(5),
c = rlnorm(5),
d = rlnorm(5)
)
df
```


Mientras que si modificamos el encabezado haciendo **{r, eval=TRUE}**

Obtendremos tanto r**esultado como código** en el documento final

```
df <- tibble::tibble(
a = rlnorm(5),
b = rlnorm(5),
c = rlnorm(5),
d = rlnorm(5)
)
df
```

a	b	c	d
<dbl>	<dbl>	<dbl>	<dbl>
0.2468128	0.3557732	0.4302330	1.7676965
0.7987009	0.2578780	2.1004538	0.6735496
1.6410585	0.3033370	1.1350614	1.4979129
0.4621796	1.0801330	0.7939378	0.7447739
2.9798967	0.5650802	5.6122074	0.8903971

5 rows

Finalmente podemos mostrar sólo el resultado de la ejecución del código, haciendo que el encabezado sea así **{r, echo=FALSE}**

Obtendremos este resultado

a	b	c	d
<dbl>	<dbl>	<dbl>	<dbl>
0.10893011	1.5215756	3.2582664	1.5945670
0.28736282	2.0715360	3.1785180	1.6095357
0.02382011	0.4306141	4.4213080	3.7623654
4.26958187	0.8683974	0.1895919	0.7304635
0.67947669	13.0387143	0.4770166	3.9702831

5 rows

Observa que en todos los casos, el código es el mismo, lo único que hemos cambiado ha sido el encabezado que acompaña al *Chunk*

4.2 Incluir tablas

Cuanto se trata de presentar los resultados en un *data frame* o tibble, de manera automática **R** muestran los resultados con un formato sencillo. Por ejemplo, escribimos esto en *RMarkdown* y generamos el documento final, obtendremos algo así.

~~~~
```{r, echo=FALSE}
babynames::births[1:15,]
```
~~~~

Obtendremos esto

```
## # A tibble: 15 x 9
##     x     qx    lx       dx      Lx      Tx       ex    sex  year
##   <dbl> <dbl> <dbl>    <dbl>   <dbl>   <dbl>    <dbl> <fct> <dbl>
## 1  0   0.146   100000   14596   90026   5151511   51.5  M     1900
## 2  1   0.0328  85404    2803    84003   5061484   59.3  M     1900
## 3  2   0.0163  82601    1350    81926   4977482   60.3  M     1900
## 4  3   0.0105  81251    855     80824   4895556   60.2  M     1900
## 5  4   0.00875 80397    703     80045   4814732   59.9  M     1900
## 6  5   0.00628 79693    501     79443   4734687   59.4  M     1900
## 7  6   0.00462 79193    366     79010   4655244   58.8  M     1900
## 8  7   0.00326 78827    257     78698   4576234   58.0  M     1900
## 9  8   0.00256 78569    201     78469   4497536   57.2  M     1900
## 10 9   0.00203 78368    159     78288   4419068   56.4  M     1900
## 11 10  0.00211 78209    165     78127   4340779   55.5  M     1900
## 12 11  0.00217 78044    169     77960   4262653   54.6  M     1900
## 13 12  0.00212 77875    165     77793   4184693   53.7  M     1900
## 14 13  0.00239 77710    186     77617   4106900   52.8  M     1900
## 15 14  0.00254 77525    197     77426   4029283   52.0  M     1900
```

El cual es un formato de una tabla sencillo. Podemos mejorar esta presentación de la tabla en el documento haciendo

~~~~
```{r, echo=FALSE}
knitr::kable(
babynames::lifetables[1:15,],
caption = "Una tabla"
)
```
~~~~

Lo cual cuando ejecutemos el documento generará, una tabla con un poco mas de calidad.

x	qx	lx	dx	Lx	Tx	ex	sex	year
0	0,14596	100000	14596	90026	5151511	51,52	M	1900
1	0,03282	85404	2803	84003	5061484	59,26	M	1900
2	0,01634	82601	1350	81926	4977482	60,26	M	1900
3	0,01052	81251	855	80824	4895556	60,25	M	1900
4	0,00875	80397	703	80045	4814732	59,89	M	1900
5	0,00628	79693	501	79443	4734687	59,41	M	1900
6	0,00462	79193	366	79010	4655244	58,78	M	1900
7	0,00326	78827	257	78698	4576234	58,05	M	1900
8	0,00256	78569	201	78469	4497536	57,24	M	1900
9	0,00203	78368	159	78288	4419068	56,39	M	1900
10	0,00211	78209	165	78127	4340779	55,50	M	1900
11	0,00217	78044	169	77960	4262653	54,62	M	1900
12	0,00212	77875	165	77793	4184693	53,74	M	1900
13	0,00239	77710	186	77617	4106900	52,85	M	1900
14	0,00254	77525	197	77426	4029283	51,97	M	1900

Al ejecutar un archivo *html* puede ser que no aprecien diferencias importantes entre ambas tablas, sin embargo, al como veremos mas adelante, es posible ejecutar otros formatos de salida, como word o pdf. En esos casos, las diferentes entre ambas tablas será muy notable.

La función *kabble* permite otra modificaciones para la presentación de una tabla en un documento final. Consultando la ayuda de esta función podrás ver la variedad de opciones que ofrece.

Recuerda que para ver todos estos resultados debes utilizar un documento *RMarkdown* y generar un HTML, ya que si los ejecutas en un *script* simple o en la consola, no podrás observar el efecto que tienen las diferentes opciones sobre los resultados mostrados.

4.3 Opciones Globales

R considera opciones globales predeterminadas para un Chunk, podrás darte cuenta de esto, cuando ejecutas el código y no hay ninguna opción activada en su encabezado. Por ejemplo, de manera automática, R siempre muestra, el código, los resultados, los warnings y los mensajes de un chunk. Si es del interés cambiar esta configuración para TODOS los chunks del documento, pues usar la opción knitr::opts_chunk$set(). Por ejemplo, para mostrar sólo los resultados, debes configurar el primer chunk del documento;

````
```{r,include=FALSE}
knitr::opts_chunk$set(
echo = FALSE
)
```
````

Recuerda que si fijas está opción al inicio de tu documento *RMarkdown* todos os *chunks* tendrán esa configuración, por lo cual ya no será necesario incluirla en el encabezado.

4.4 Código en línea

Hemos dicho que en un documento tipo *RMarkdown* debemos insertar un *Chunk* para indicar que en ese espacio deseamos incluir código y no texto. En ocasiones es necesario, incluir pequeñas líneas de código que acompañen el texto. Por ejemplo; si tenemos una muestra de datos, y queremos referirnos a su tamaño, podemos referirnos a el, usando texto común en combinación con lo que se llama *code line* o código en línea.

De manera que si escribimos esto;

```
- La muestra total contiene `r dim(babynames::births)` filas y columnas
```

Veremos en el documento final, esto

- La muestra total contiene 109, 2 filas y columnas.

Podemos hacer operaciones haciendo uso de código en línea de manera similar. Por ejemplo, si hacemos esto en el *RMarkdown*

```
- De `r babynames::births[1,1]` a `r babynames::births[nrow(babynames::bir-
ths),1]`   se registraron `r sum(babynames::births$births)` nacimientos en
Estados Unidos.
```

Veremos en el documento final esto

- De 1909 a 2017 se registraron 379185524 nacimientos en Estados Unidos.

Cuando usamos código en línea **R** nos regresará el valor numérico contenido en una variable sin formato alguno. Para una mejor presentación podríamos construir una función. Por ejemplo, si hacemos

````
```{r, include=FALSE}
coma<- function(x) format(x, digits = 2, big.mark = ",")
```
````


Podríamos obtener un mejor resultado para la expresión anterior

```
- De `r babynames::births[1,1]` a `r babynames::births[nrow(babynames::bir-
ths),1]`   se registraron `r coma(sum(babynames::births$births))` nacimientos
en Estados Unidos.
```

- De 1909 a 2017 se registraron 379,185,524 nacimientos en Estados Unidos.

5 Encabezado YAML

En el encabezado de un *RMarkdown* es posible controlar configuraciones de todo el documento. Por ejemplo, el documento generado de manera automática por **R** cuando seleccionamos un archivo nuevo de tipo *RMarkdown* tiene un encabezado específico el cual, por lo general, incluye lo siguiente:

```
---
title: "Untitled"
author: "Yo"
date: "1/24/2021"
output: html_document
---
```

Este encabezado se puede modificar con el fin de cambiar el documento en general. Empezaremos trabajando para modificar ciertos parámetros del documento y añadir numeración a los diferentes componentes.

5.1 Paramétros

Un archivo de tipo RMarkdown puede incluir uno o mas parámetros. Un parámetro se refiere a un determinado objeto con el cual deseamos que **R** trabaje en la ejecución. Si ese objeto cambia, **R** automáticamente actualizará los resultados dependiendo del parámetro indicado. Para indicar que deseamos trabajar con parámetros, debemos incluir la opción **params:** seguida de los parámetros que vamos a utilizar en el encabezado del documento.

Usemos los datos de *mibici* para ejemplificar el uso de parámetros. Supongamos que deseamos construir un reporte en el cual se considere únicamente el sexo de las personas (hombre o mujer) y un día especifico del mes de diciembre 2020. Con estos datos construyamos, un documento que muestre la duración promedio de los viajes en ese día para ese género y una gráfica con la duración promedio de los viajes iniciados en determinada hora. Esta base ya la trabajamos en el capítulo 9, por lo que usaremos los resultados previamente obtenidos y no entraremos en el detalle de la construcción de gráficas.

Comencemos creando un documento *RMarkdown*, tal y como lo explicamos anteriormente. Definamos los parámetros en el encabezado. En este caso hay dos, que son los valores que deseamos cambiar; uno el sexo y el otro del día del mes de diciembre. En los parámetros indicamos un valor de referencia.

```
---
title: "Viajes en Mi Bici"
output: html_document
params:
sexo: "M"
dia: 12
---
```

Se copias y pegas este código asegúrate que se respete la indentación del código.

Después que establecemos esos parámetros, definimos quien será la base de datos con lo cual trabajaremos. En este caso, *mibici,* por lo cual la cargamos como de costumbre, incluyendo las librerías que necesitaremos. En el siguiente código hemos cargado la librería e hicimos un filtro de nombre *mostrar* en el cual se filtrarán los datos donde el género sea igual a lo que se almacena en el parámetro sexo indicado por **params$sexo** y el día se igual a lo que se especifica en el parámetro día indicado por **params$dia**.

```
```{r, message=FALSE, warning=FALSE, include=FALSE}
library(readr)
library(dplyr)
library(lubridate)
library(ggplot2)
setwd("~/Dropbox/Curso de R/Cap16_Markdown")
mibici<-read_csv("mibici.csv")
mostrar <- mibici %>%
filter(genero==params$sexo, day==params$dia)
```
```

El filtro efectuado define el conjunto de datos sobre el cual se trabajará la información. Lo que resta hacer es definir el proceso de datos para obtener el promedio de la duración de los viajes por hora, para el día seleccionado.

```
```{r, exercise = TRUE}
mibici2<-mostrar %>%
     mutate(inicio=make_datetime(year, month, day, hour_inicio, minute_ini
     cio, second_inicio))%>%
     mutate(final=make_datetime(year, month, day, hour_final, minute_final,
```



```
     second_final))%>%
     mutate(dura=final-inicio)
     mibici2[1:5,16:18]
```
```

Con estos elementos, podemos escribir la siguiente información que deseamos en el documento como;

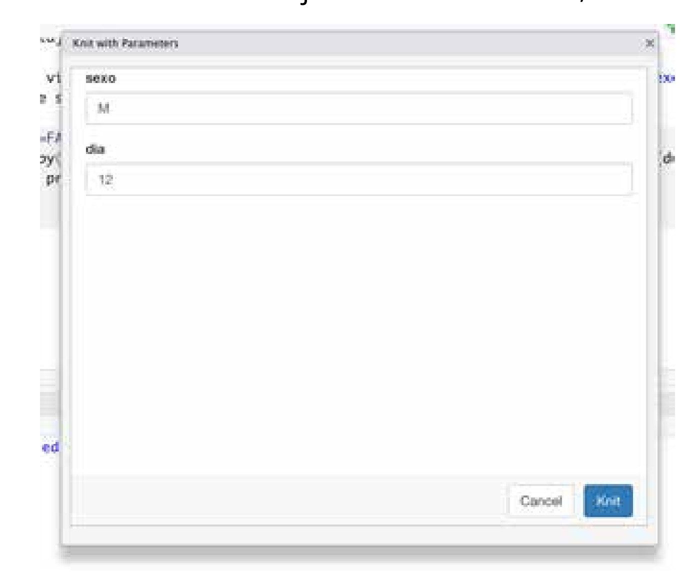
```
Viajes efectuados por personas de sexo **`r params$sexo`** en el día **`r
params$dia`** del mes de diciembre del año 2020. La duración promedio de los
viajes en este día fue de **`r as.numeric(mean(mibici2$dura))`** segundos.
```

Finalmente incluimos la gráfica

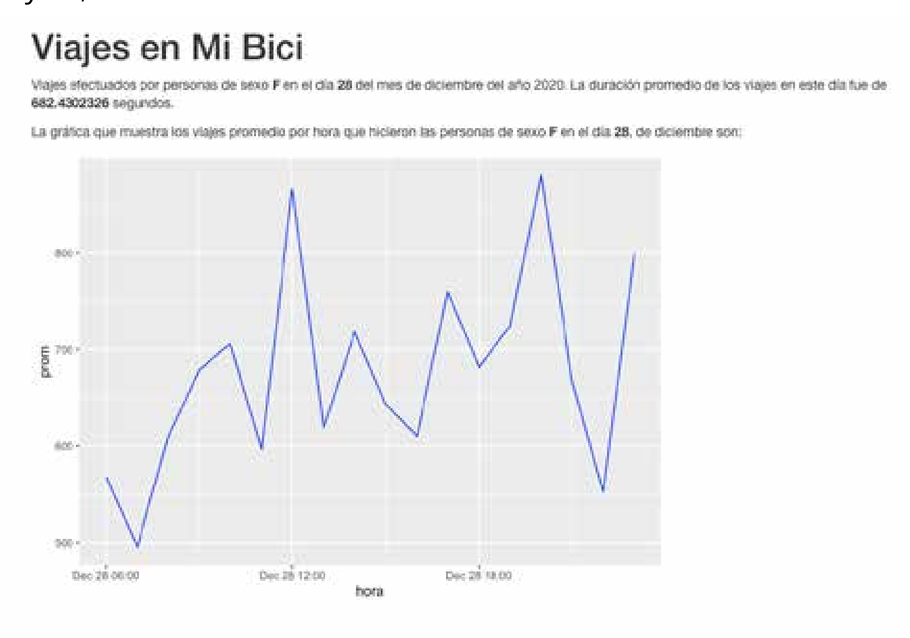
```
La gráfica que muestra los viajes promedio por hora que hicieron las personas
de sexo **`r params$sexo`** en el día **`r params$dia`**, de diciembre son:
```

```
```{r, message=FALSE, echo=FALSE}
promedio<-summarise(group_by(mibici2, hora=make_datetime(year, month, day,
hour_inicio)),
prom=mean(dura), .groups = 'drop')
ggplot(promedio, aes(hora, prom))+
     geom_line(col="blue")
```
```

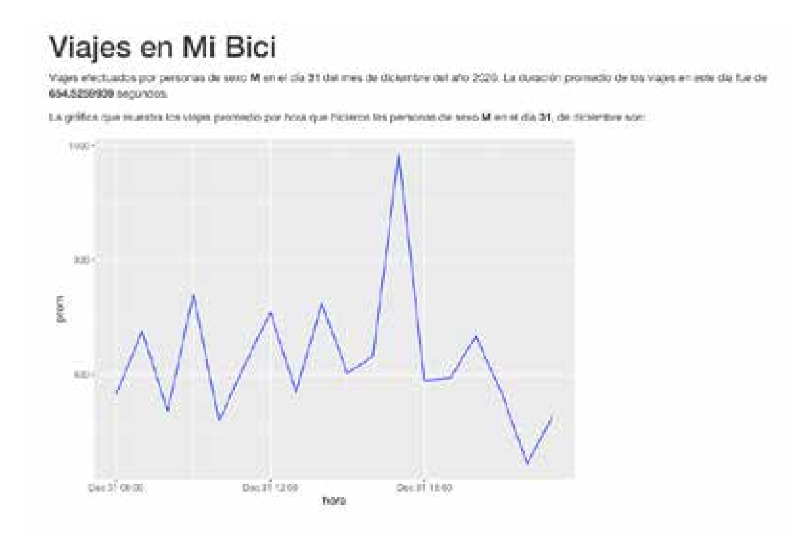
Observa que hemos definido algunas características en el encabezado del *chunk*. Cuando incluimos parámetros en un documento *RMarkdown*, la forma de generar el documento final es mediante el uso de la opción *Knit with parameters* . Guarda esta documento con el nombre *MiBici.Rmd* **ya que nos referiremos a el mas adelante**. Al ejecutarlo tendremos;



Donde podemos indicar el valor que deseamos para cada uno de los parámetros. Obviamente los parámetros que indiquemos deben ser valores posibles de las variables incluidas en los datos. En este caso, sexo solo puede tomar dos valores M o F, mientras que día, los números del 1 al 31. Si seleccionamos F y 28, obtenemos



Este documento se abrirá en una pestaña alterna de un navegador como *html.* Si volvemos a ejecutar cambiando los parámetros seleccionando sexo M y día 31, tendremos algo así.



Otra alternativa para ejecutar el *Rmarkdown* usando parámetros es usar la función render directo de *rmarkdown*

Abre un nuevo *script* y efectúa lo siguiente;

```
setwd("~/Dropbox/Curso de R/Cap16_Markdown")
rmarkdown::render(
"MiBici.Rmd",
```



```
params = list(sexo = "M", dia=12 ),
output_file = stringr::str_c("Rep_MiBici_", "M", "12", ".html")
)
```

Con esta opción en el directorio indicado hemos guardado el archivo *html*.

De esta manera si tienes un reporte dinámico, es posible ejecutarlo desde un *script* y esto se vuelve útil si tienes diferentes tipos de reportes que desees ejecutar al mismo tiempo.

Podemos crear también una función para ejecutar el documento *html* con diferentes parámetros. Para ello debemos ser muy cuidadosos, pues el archivo *Rmd* a ejecutar debe estar contenido dentro del mismo directorio.

Prueba con la siguiente función desde un *script*.

```
hacer_reporte_MiBici <- function(sexo, dia){
   rmarkdown::render(
     "Mibici.Rmd", params = list(sexo=sexo, dia=dia),
     output_file = stringr::str_c("Rep_MiBici_", sexo, dia, ".html"))
}
```

De esta manera podemos hacer directo el reporte simplemente haciendo;

```
hacer_reporte_MiBici("F", 30)
```

Recuerda que para que esto funcione el llamado de esta función debe hacerse fuera del documento *MiBici* y la base de datos, debe estar contenida en el mismo directorio.

Con la función que hemos creado podríamos automatizar la generación de reportes usando *pwalk* de la librería *purrr*. Primero tendríamos que escribir las combinaciones deseadas

```
parametros <- list(sexo=c("F", "M", "F"), dia=c(1,5,20))
parametros
```

```
## $sexo
## [1] "F" "M" "F"
##
## $dia
## [1]  1  5 20
```

En este caso ejecutaremos los reportes F-1, M-5, F-20.

```
purrr::pwalk(parametros, hacer_reporte_MiBici)
```

Si fuese de nuestro interés generar **TODOS** los reportes posibles con estos parámetros, tendría-

mos que hacer lo siguiente para obtener las mas de 60 posibilidades y después ejecutar nuevamente *pwalk*

```
parametros <- list(rep(unique(mibici$genero),31), rep(unique(mibici$day), 2))
parametros
```

```
## [[1]]
## [1] "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M"
## [20] "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F"
## [39] "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M" "F" "M"
## [58] "F" "M" "F" "M" "F"
##
## [[2]]
## [1] 22 23 15 6 9 19 21 14 5 16 2 28 31 4 17 18 30 25 3 13 7 1 20 10 24
## [26] 29 11 8 26 12 27 22 23 15 6 9 19 21 14 5 16 2 28 31 4 17 18 30 25 3
## [51] 13 7 1 20 10 24 29 11 8 26 12 27
```

Esta idea de generación automática de reportes generar mayor eficiencia en el trabajo y reduce la posibilidad de errores.

5.2 Numeraciones y temas

Cuando trabajamos en un documento *RMarkdown* podemos escribir tanto texto como deseemos, también incluir diferentes secciones y apartados para que el trabajo que desarrollemos se vea ordenado y sea mas claro. Vuelve a abrir el archivo *MiBici.Rmd* y modifica el *encabezado* del documento.

```
---
title: "Viajes en Mi Bici"
output:
html_document:
toc: yes
number_sections: yes
params:
sexo: M
dia: 12
---
```


Para ver el funcionamiento agreguemos esto al final sólo con fines visuales.

```
# Introducción

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse a euis-
mod neque.
## Requisitos

Curabitur laoreet lectus eget tempus lacinia.
# Capítulo Primero

Morbi felis nisi, auctor sed dolor vel, consequat aliquam mauris.
```

Con estas modificaciones en la parte superior del documento tendremos una tabla de contenidos. La cual puedes visualizar el ejecutar el *html*.

Viajes en Mi Bici

- 1 Introducción
  - 1.1 Requisitos
- 2 Capítulo Primero

Viajes efectuados por personas de sexo **M** en el día **12** del mes de di

Y podremos apreciar una numeración

1 Introducción

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse a euismod neque. Cras vestibulum, erat nec lobortis lacinia, arcu lectus tincidunt turpis, ac ultricies elit arcu quis erat. Nam convallis posuere odio, in auctor neque sodales id. Mauris eu viverra diam, vitae pharetra ex.

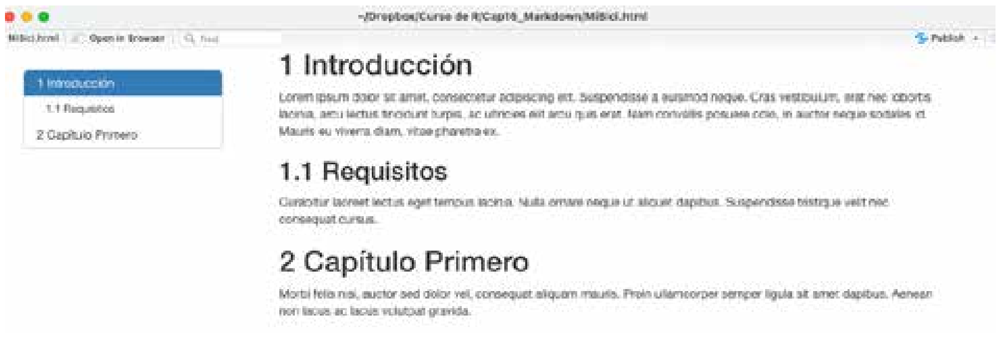
1.1 Requisitos

Curabitur laoreet lectus eget tempus lacinia. Nulla ornare neque ut aliquet dapibus. Suspendisse tristique velit nec consequat cursus.

2 Capítulo Primero

Morbi felis nisi, auctor sed dolor vel, consequat aliquam mauris. Proin ullamcorper semper ligula sit amet dapibus. Aenean non lacus ac lacus volutpat gravida.

Si en el encabezado agregamos **toc_float: yes** al ejecutar el *html* podremos observar una tabla de contenidos que permite un mejor desplazamiento sobre el documento.



6 Otros Formatos de RMarkdown

Rmarkdown permite generar diferentes tipos de formatos alternos a documentos *html.* Por ejemplo, en la opción *Knit* se presenta *Knit to word* lo que generará el resultado del documento en un archivo de texto word que puede manejarse como en procesador de texto.

Para ejecutarlo cambiamos el tipo de documento y la extensión. Prueba con este código en un *script* o desde la consola.

```
setwd("~/Dropbox/Curso de R/Cap16_Markdown")
rmarkdown::render(
"MiBici.Rmd",
params = list(sexo = "M", dia=12 ),
output_file = stringr::str_c("Rep_MiBici_", "M", "12", ".docx"),
output_format = "word_document"
)
```

Para incluir mas especificaciones sobre la salida de un reporte, podemos modificar el encabezado del documento.

Elaborar un *RMarkdown* permite una amplia gama de posibilidades para generar salidas de reportes. En algunos casos, debemos instalar herramientas adicionales para poder obtener un documento en el formato deseado.

6.1 Documentos

Algunos otros de los documentos de salida, pueden ser del tipo

- **pdf_document** el cual generará un documento *.pdf* mediante el uso de *LaTex*, lo cual requiere tener instalado este programa en computadora.
- **word_document** generará un documento de con word con extensión *.docx*
- **odt_document** generará un documento de texto abierto con extensión *.odt*
- **rtf_document** un documento de texto enriquecido, con formato *.rtf*
- **md_document** un documento *Markdown*

La información visible en el documento final dependerá de las indicaciones que se hayan establecido sobre los diferentes *chunks*. Por ejemplo, en los encabezados el documento de *MiBici* cambia la opción **include=FALSE** por **include=TRUE** e incluye lo siguiente en el encabezado del documento **code_folding: hide**.

```
---
title: "Viajes en Mi Bici"
output:
```



```
html_document:
toc: yes
toc_float: yes
number_sections: yes
code_folding: hide
params:
sexo: M
dia: 12
---
```

Con estos cambios genera el *html* y observa que ahora aparece la operación de un botón de nombre *code.* Esto permite ocultar el código a menos que se seleccione con este botón.



6.2 Notebooks

Presentar un análisis de resultados generando un *html* resulta de gran ayuda porque te permite presentar la información para que las personas puedan tomar decisiones de manera informada. Si el objetivo mas que presentar resultados es permitir que otros exploren tu trabajo, hagan modificaciones y mejoras, un documento *html, word* o *pdf* no será lo mas adecuado. En estos casos conviene el uso de lo que se conoce con un *Notebook*, con el cual, ademas de tener acceso a código, texto y resultados se incluye el archivo *.Rmd*.

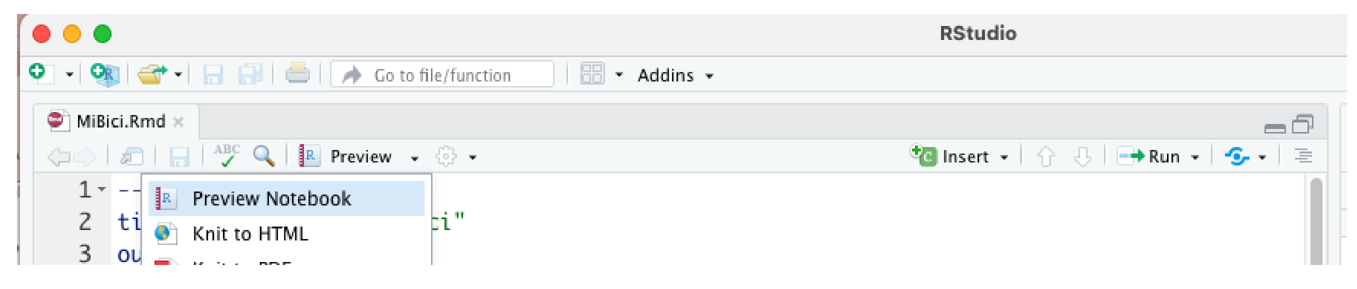
Un *Notebook* tiene una extensión *.nb.html* y para generarlo tenemos dos opciones. Si ya tenemos un *RMarkdown*, únicamente debemos modificar el encabezado agregando la opción **html_notebook: default**. Si vamos a empezar de cero, podemos seleccionarlo desde el menú *File, NewFile, Notebook*

Modifica el encabezad del archivo *MiBi.Rmd* de la siguiente manera;

```
---
title: "Viajes en Mi Bici"
output:
   html_notebook: default
   html_document:
```

```
    toc: yes
    toc_float: yes
    number_sections: yes
    code_folding: hide
params:
  sexo: M
  dia: 12
---
```

Cuando incluimos esta opción en el documento te darás cuenta que la opción *Knit* cambia y muestra una opción *Preview*.



Al seleccionarla y ejecutar un *Preview Notebook* obtenemos una pre-visualzación que nos muestra la opción de descargar el archivo *.Rmd*

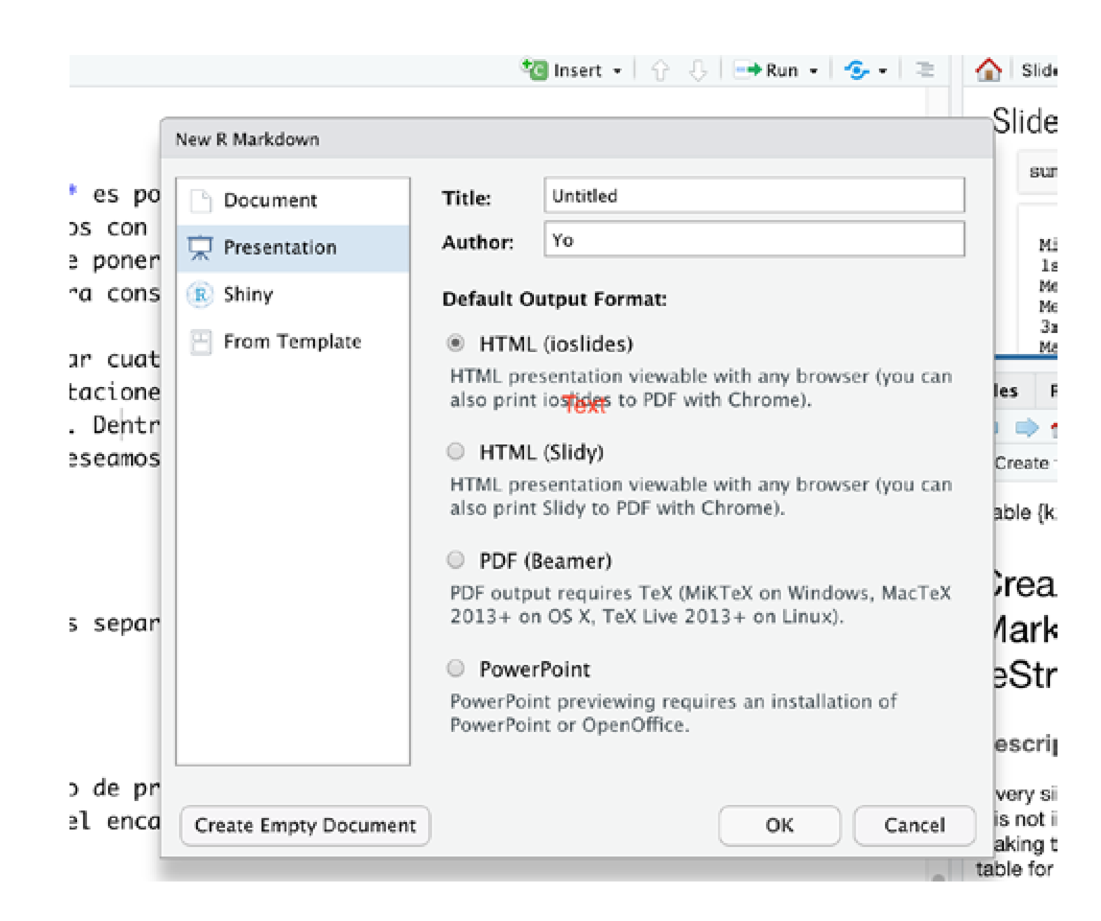


En nuestro directorio tenemos ahora un archivo de nombre *MiBici.nb.html* que podemos compartir, en el cual, se incluye además la posibilidad de descargar el *Rmd* que lo acompaña.

6.3 Presentaciones

Por medio de un *RMarkdown* es posible también construir presentaciones. Esta herramienta te permite poner tus resultados en diapositivas, en lugar de estar copiando y pegando de manera constante.

En *RMarkdown* podemos crear cuatro tipos de presentaciones. Para trabajar con los diferentes tipos de presentaciones podemos seleccionar *File, New File, RMarkdown* y seleccionar *Presentation.* Dentro de las opciones podemos indicar el tipo de presentación con la cual deseamos trabajar.



Cualquiera que sea el tipo de presentación que seleccionemos **R** generará al igual que antes, un archivo automático de prueba, con cierta información genérica que nos da una idea sobre las posibilidades.

- Presentaciones ioslides

Este tipo de presentación se incluye dentro del encabezado del R*Markdown* la opción **output: ioslides_presentation**. Para insertar dispositivas, usamos # y ## dependiendo si se trata de una dispositiva de titulo o de contenidos.

Para insertar código y resultados en una diapositiva, usamos los mismos principios para que usamos anteriormente, pues es exactamente el mismo tipo de documento, lo que hemos cambiado ahora, es la salida que deseamos observar. Genera una presentación *ioslides* y ejecuta el *knit* con la opción *Knit to html (ioslides)*

observa el documento generado.

- Presentaciones Slidy

Estas presentaciones incluyen la salida con **output: slidy_presentation**. Has el mismo ejercicio, genera una presentación de este tipo y ejecuta el *knit* con la opción *Knit* to *html (Slidy)* observa el documento generado.

- Presentaciones Beamer

Las presentaciones tipo *Beamer* generan una salida muy similar a las presentaciones construidas usando LaTex, por lo cual debes tener en tu computadora instalado este programa. Una presenta-

ción de este tipo, incluye en el encabezado **output: beamer_presentation**

- Presentaciones PPT

En este caso en el encabezado se incluye la opción **output: powerpoint_presentatio**n. Al ejecutar el *knit* con la opción *Knit* to *PowerPoint* se generar automáticamente un documento de este tipo, con la información.

Una desventaja al trabajar con estos elementos es que las dispositivas en ocasiones no se ajustan de manera automática al contenido de nuestro texto. Prueba con las cuatro opciones haciendo un documento nuevo de cada tipo, copia y pega el contenido del documento *MiBici* excepto el encabezado y observa los distintos formatos y opciones que ofrecen las presentaciones.

Todos los elementos sobre manipulación de documentos *Rmarkdown* que hemos revisado para generar *html* funcionan de la misma manera. Recuerda que la única diferencia es el tipo de salida que estamos dando al documento.

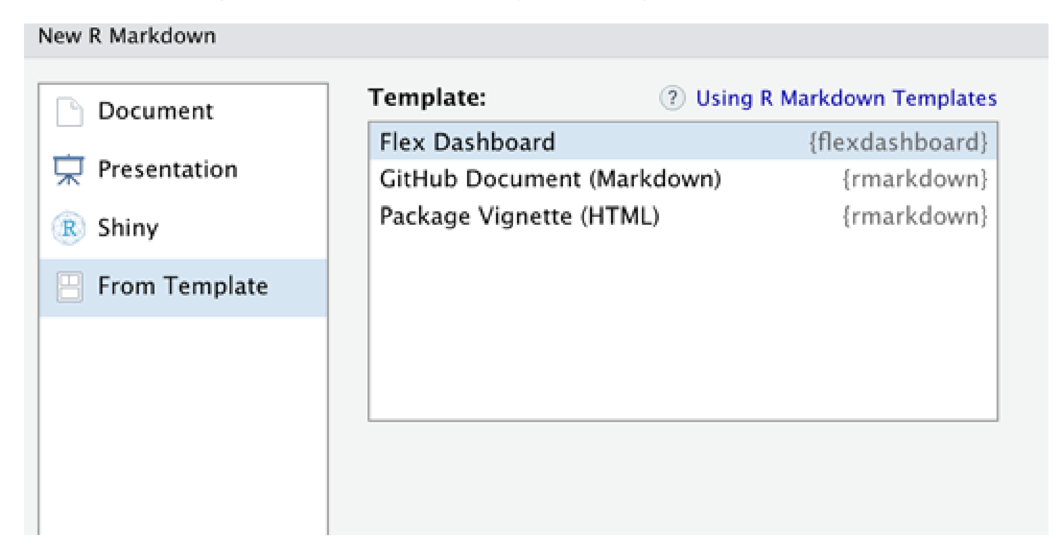
6.4 Dashboards

Las herramientas que hemos trabajado nos permiten presentar los resultados de un análisis de distintas formas. En ocasiones mas que presentar algunos resultados, nos interesa que las personas puedan ver toda la información. Los *Dashboards* son una herramienta que permite esta funcionalidad.

Un *Dashboard* es una forma alternativa de presentación usando un documento *Rmarkdown.* La salida en este caso es **output: flexdashboard::flex_dashboard**. En un *dashboard* un **#** significa una pestaña, **##** una columna dentro de la pestaña, mientras que **###** hacen una fila dentro de la columna.

Para usar *Dashboard* es necesario instalar la librería **flexdashboard**. Para generar un *Dashboard* hay dos opciones;

- Generar un archivo tipo *RMarkdown* nuevo y en Templates seleccionar *Flex Dashboard*



Crear un documento *Markdown* normal y agregar *output: flexdashboard::flex_dashboard* en ambos casos es necesario tener instalada la librería mencionada.

Para ejemplificar el funcionamiento de un *Dashboard* usaremos los datos y algunas de las gráficas previamente utilizadas en el Cap. 3. En este capítulo aprendimos a usar *ggplot2* por lo que las gráficas que usaremos se trabajaron previamente en esa parte del curso.

Usaremos la librería **DT** la cual nos permite visualizar una tabla de datos de forma mas interactiva, como lo verás en el siguiente *dashboard*. Instala antes de continuar.

Prepara un documento para el **Dashboard** usando una de las opciones anteriores, copia y pega lo siguiente, **No olvides modificar el directorio donde este la base de datos para que sea posible utilizarla**.

```
---
title: "Mi Primer Dashboard"
author: "Yo"
date: "1/27/2021"
output:
     flexdashboard::flex_dashboard:
          social: menu
          source_code: embed
---


# Ingresos y años de escolaridad
```{r}
library(readxl)
library(tidyverse)
library(DT)
```

```{r}
setwd("~/Dropbox/Curso de R")
enoe<-read_xlsx("mu_enoe.xlsx")
```
Columna 1

Ingreso vs Años escolaridad

```{r}
ggplot(data = enoe) +
geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual ))```

### Diferencias en el salario promedio

```{r}
```

````
ggplot(data=enoe)+
gbar man(mapping = aes(x=sex, fill=sex))
```

## Columna 2

```{r}
enoe %>%
arrange(desc(ingreso_mensual)) %>%
head(500) %>%
select(estado, sex, edad, niv_edu, ingreso_mensual) %>%
DT::datatable()
```

# Nivel educativo

## Columna 1

### Información por sexo

```{r}
ggplot(data=enoe)+
geom_bar(mapping = aes(x=sex, fill=niv_edu))
```

### Ubicación

```{r}
library(leaflet)
leaflet() %>%
setView(-103.38742861607173, 20.67460296566588, zoom = 16) %>%
addTiles() %>%
addMarkers(-103.38742861607173, 20.67460296566588, popup = "Minerva")
```

## Columna 2

### Observaciones

```{r}
ggplot(data=enoe, mapping = aes(x=sex, fill=niv_edu))+
geom_bar( position = "dodge")+
labs(title="Observaciones por sexo y nivel educativo", x="Sexo", y="Observa-
ciones")
```

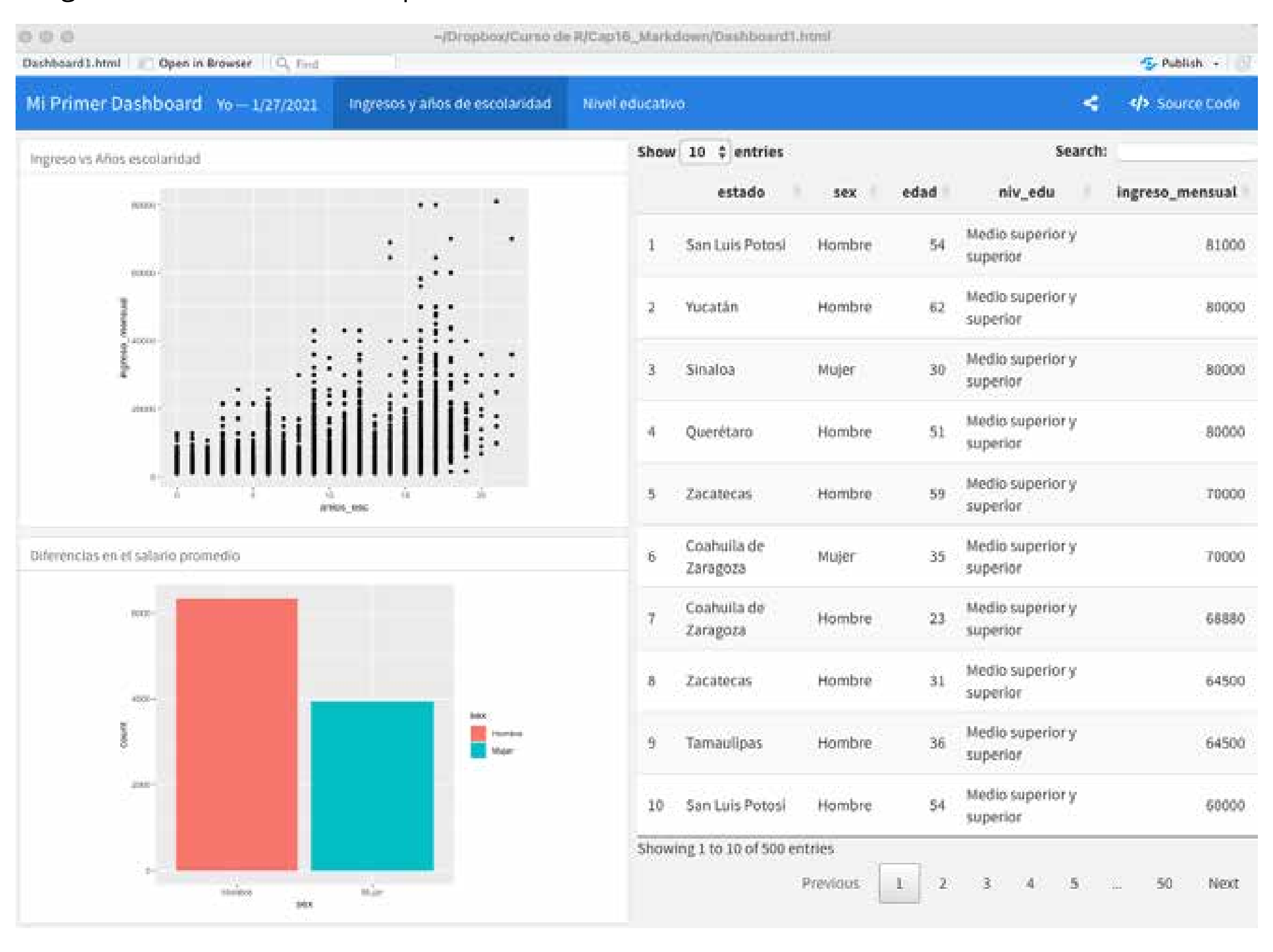
### Diagrama caja
````



````
```{r}
ggplot(data=enoe, mapping = aes(x=niv_edu, y=ingreso_mensual))+
geom_boxplot()+
coord_flip()
```
````

Guarda el documento como un archivo *.Rmd* de nombre *Dashboard*1. Una vez que lo has guardado, veras que en el *Knit* ahora tienes la opción *Knit to flex_dashboard* en caso de que la opción no te aparezca cierra el documento y vuelve a abrirlo. Ejecuta *Knit to flex_dashboard*

Esto, generará un documento que luce mas o menos así

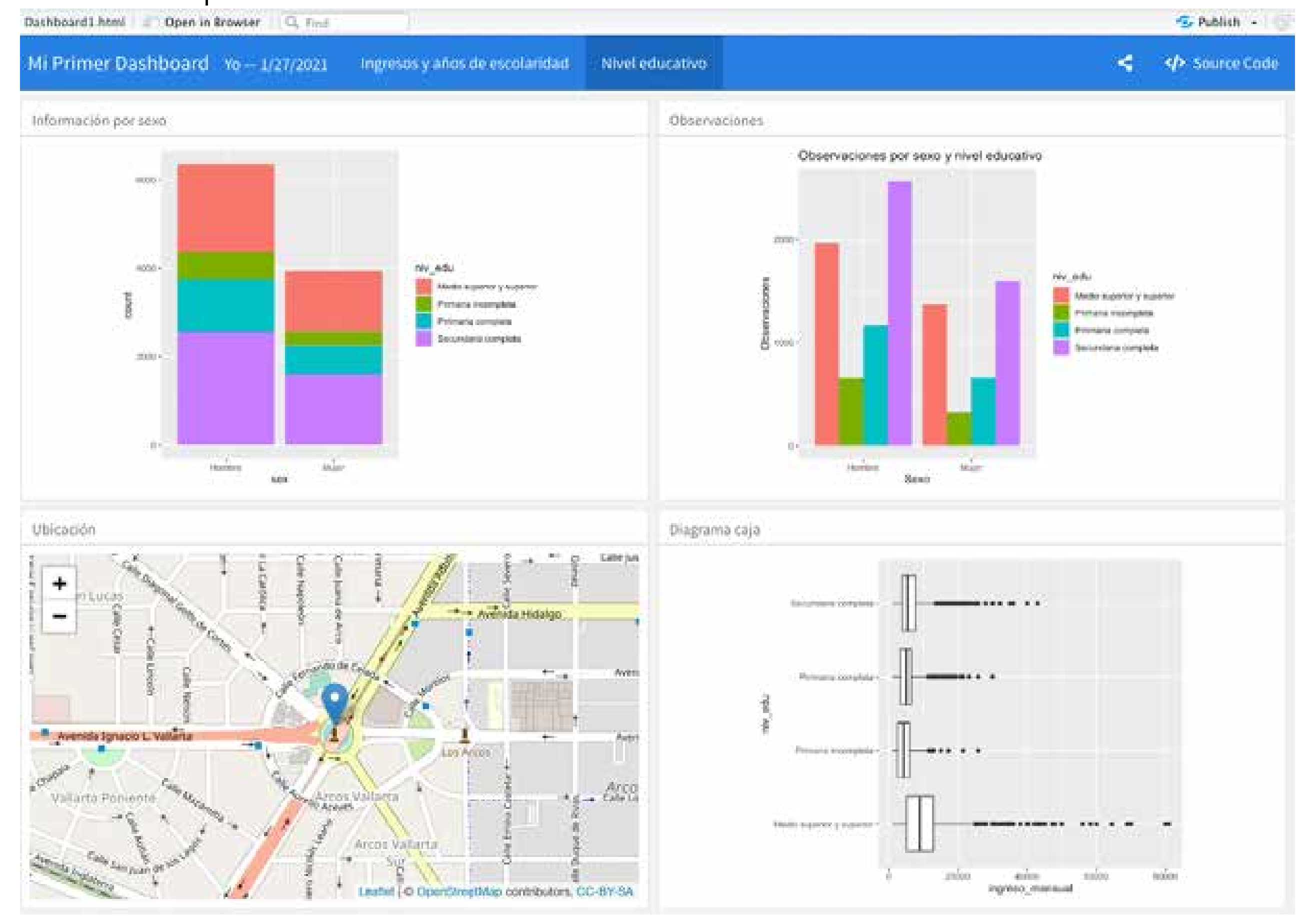


Este documento tiene dos pestañas, en la primera hay dos columnas y la primera columna tiene dos filas. Esto resulta de la indicación usando #, ## o ###. Observa que en la parte superior derecha del *Dashboard* hay una opción que nombre *Source Code*. Al presionarla permite ver el código con el cual se generó. Estos es porque hemos incluido la opción **source_code: embed.** El símbolo compartir aparece porque seleccionamos **social: menu.**

La tabla que aparece del lado derecho de la primera pestaña se encuentra en un formato por medio del cual es posible desplazarse entre los datos.

La segunda pestaña contiene algunas gráficas que ya hemos desarrollado antes. Se incluye tam-
````

bién en la parte inferior derecha la ubicación de un punto de referencia de la ciudad de Guadalajara. Esta es generada gracias a la librería **leaflet** donde lo único que hicimos fue añadir las coordenadas de un punto.



Tanto la ubicación como la tabla son funcionalidades *html* que *R* permite sin necesidad de que sepam programar en lenguajes como JavaScript.

Las posibilidades que ofrece un *Dashboard* son muy amplias. Puedes consultar mas ejemplos e ideas para elaborarlos aquí FDL

6.5 Shiny

Si bien el *Dashboard* que construimos con anterioridad permite mostrar información mas completa, no es interactivo. Por ejemplo, quisiéramos que quien lo usa tenga un menú en el que pueda decidir que tipo de información mostrar. Esto es posible gracias al uso combinado de este *Dashboard* con una librería que ofrece mayores posibilidades, conocida como *shiny*. Para ver sólo un poco de como podemos usarla, modifiquemos el documento *Dasboard1*. Las modificaciones que efectuaremos incluyen;

- Agregar runtime: shiny en el encabezado del documento. Esto permite indicar que se trata de un documento que se ejecutará a traves de shiny de forma reactiva.

- Incluir dentro de la primera pestaña un menú, donde se muestre un listado de estados. Para ello usamos

```
selectInput("Estado",      label      =      "Selecciona      el      estado:,"
choices = unique(enoe$estado), selected = "Jalisco")
```

La selección que efectuada se guardará dentro de la variable *Estado*. Para referirnos a ella, dentro del entorno de este documento interactivo debemos usar la expresión **input$Estado**.

- Incluir una base que sea reactiva, es decir que responda a la selección del estado que haga el usuario. Esto se logra incluyendo

```
enoe2←reactive(filter(enoe, estado==input$Estado))
```

- Finalmente en cada gráfica que deseamos que cambie con la selección, debemos incluir la opción

```
renderPlot({ })
```

No olvides que estamos trabajando con una base de datos, por tanto debemos asegurarnos de cargarla correctamente, ya sea a través de fijar el directorio o incluirlo al leerlo.

Modificando estos elementos **sólo en la primera pestaña,** tendremos el siguiente código

~~~~
---
title: "Mi Primer Dashboard Interactivo"
author: "Yo"
date: "1/27/2021"
output:
      flexdashboard::flex_dashboard:
            social: menu
            source_code: embed
runtime: shiny
---

# Ingresos y años de escolaridad
```{r}
library(readxl)
library(tidyverse)
library(shiny)
library(DT)
```

```{r}
setwd("~/Dropbox/Curso de R")

enoe<-read_xlsx("mu_enoe.xlsx")
```

## Columna 1

```{r}
selectInput("Estado", label = "Selecciona el estado:",
                 choices = unique(enoe$estado), selected = "Jalisco")
```

```{r}
enoe2<-reactive(filter(enoe, estado==input$Estado))
```

### Ingreso vs Años escolaridad

```{r}
renderPlot({
ggplot(enoe2()) +
geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual ))
})
```

### Diferencias en el salario promedio
~~~~

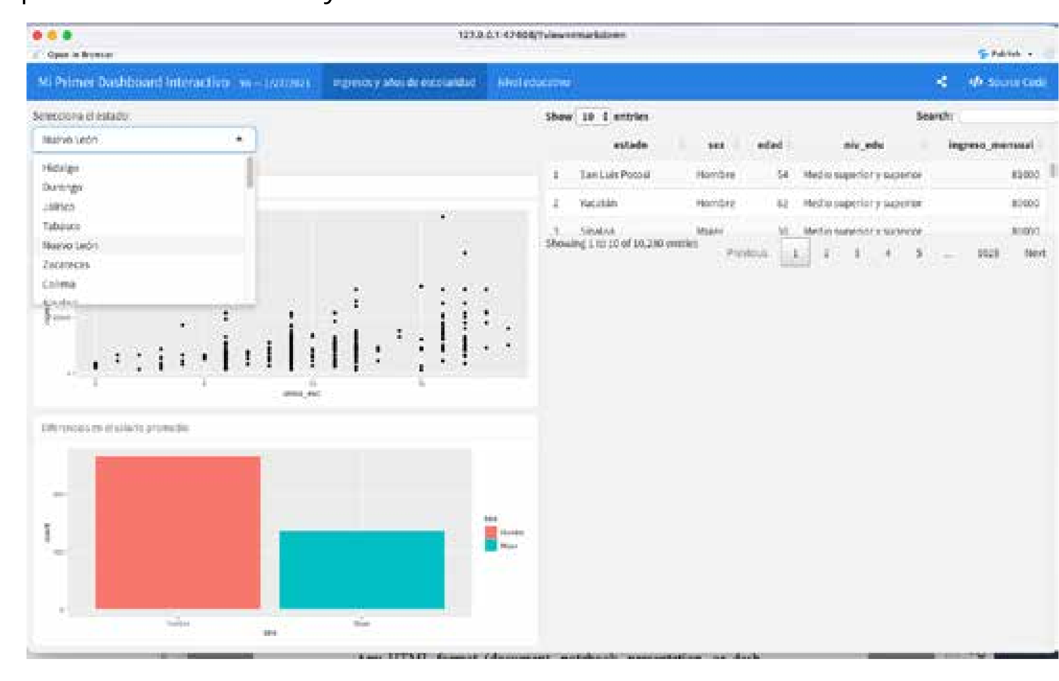


~~~~
```{r}
renderPlot({
ggplot(enoe2())+
geom_bar(mapping = aes(x=sex, fill=sex))
})
```

## Columna 2

```{r}
enoe %>%
arrange(desc(ingreso_mensual)) %>%
select(estado, sex, edad, niv_edu, ingreso_mensual) %>%
DT::datatable()
```
~~~~

Al ejecutar el *Knit to Dashboard* obtendremos algo mas o menos así. Observa que ahora en este caso, hay una lista desplegable que incluye los estados posibles. Únicamente las dos primeras gráficas de la pestaña son reactivas y cambian con la selección del estado.



6.6 Otros formatos

R ofrecen muchas mas funcionalidades semejantes a las que hemos presentado en este capítulo. Únicamente hemos presentado los elementos básicos. La librería shinny ofrece una amplia gama de posibilidades para desarrollar aplicaciones interactivas las cuales pueden montarse en un sitio y consultarse en línea. Puedes profundizar acerca de ello aquí. SH

7 Actividades

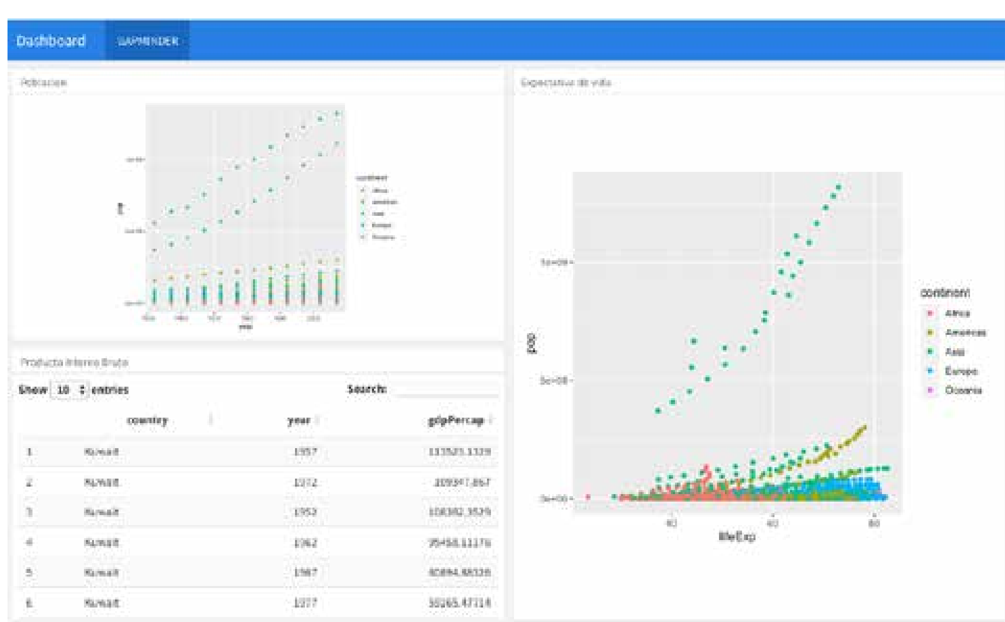
1. Crea un nuevo archivo tipo *RMarkdown*. Insertar un chunk y sobre el, escribe el siguiente código. Modifica los encabezados del chunk para que sólo sea posible observar el resultado de la ejecución

```
library(tidyverse)
library(gapminder)
gapminder %>%
ggplot()+
geom_point(aes(year, pop, col=continent))
```

2. Crea la siguiente estructura en un archivo RMarKdown

- 1 Introducción
  - 1.1 Conociendo la base de datos
  - 1.2 Especificaciones
- 2 El principal Ingreso
  - 2.1 Salarios
  - 2.2 Negocios
- 3 El principal gasto
  - 3.1 Comida
  - 3.2 Educación

3. Usa la librería gapminder para obtener el conjunto de datos contenidos en ella y generar el siguiente dashboard



Capítulo 17

Redux con ggplot2

Capítulo 17 | Redux con ggplot2

1 Introducción

En capítulos anteriores aprendimos a manipular y transformar datos, aprendimos también a hacer gráficas y representar de manera visual los datos que deseamos comunicar. Los gráficos desarrollados anteriormente no tenían ninguna especie de formato y su configuración era únicamente la determina de forma automática por **ggplot2**. En este capítulo nos enfocaremos en revisar algunas de las funciones mas comunes que permiten cambiar entre otras cosas el color, el tema de nuestros gráficos. Para mostrar este funcionamiento usaremos, la base de datos *enoe,* que fue con la que aprendimos a hacer gráficos y tomaremos uno de ellos, para ver las diferentes modificaciones posibles.

2 Previos

Utilizaremos la siguientes librerías;

```
library(readxl)
library(tidyverse)
library(ggrepel)
library(gridExtra)
library(lubridate)
```

Encontramos dos librerías nuevas **gridExtra** y **ggrepel**. La primera de ellas nos permitirá mostrar gráficas juntas con el uso de la función **grid.arrange().** Mientra que la segunda contiene algunos elementos que potencializan las gráficas que se elaboran con **ggplot2**.

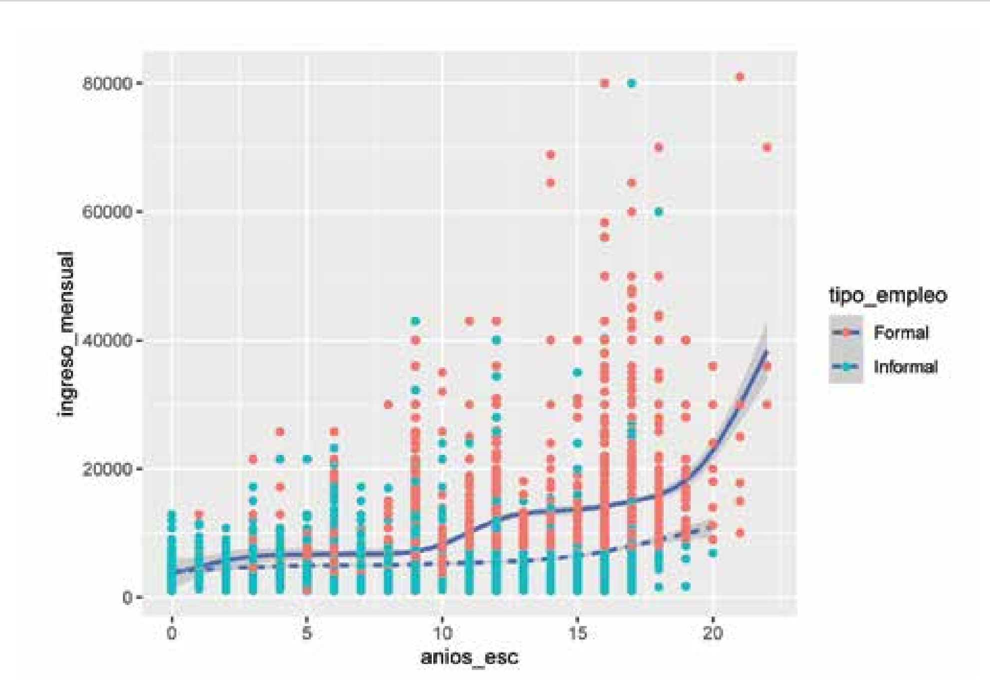
3 Etiquetas

Comencemos cargando la base de datos, en la que tenemos la información. Usaremos la muestra de le *enoe* con la que habíamos trabajado previamente. La finalidad de trabajar con la misma base es que podamos mejorar algunas de las gráficas que habíamos efectuado previamente. Como siempre definamos el directorio donde se encuentra la base que utilizaremos.

```
setwd("~/Dropbox/Curso de R/Cap17_Redux_ggplot2")
enoe←read_xlsx("mu_enoe.xlsx")
```

En el capítulo tres, presentamos esta gráfica para ver la relación entre los anos de escolaridad y el ingreso mensual de las personas que contestaron la *enoe.*

```
g1 ← ggplot(data=enoe)+
geom_smooth(mapping = aes(x =anios_esc, y =ingreso_mensual, linetype=tipo_em-
pleo))+
geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, color=tipo_em-
pleo))
g1
```



Observa que hemos creado un objeto de nombre **g1** que la información de la gráfica.

```
class(g1)
```

```
## [1] "gg" "ggplot"
```

Esta forma de referirnos a una gráfica será de mucha utilidad a partir de ahora.

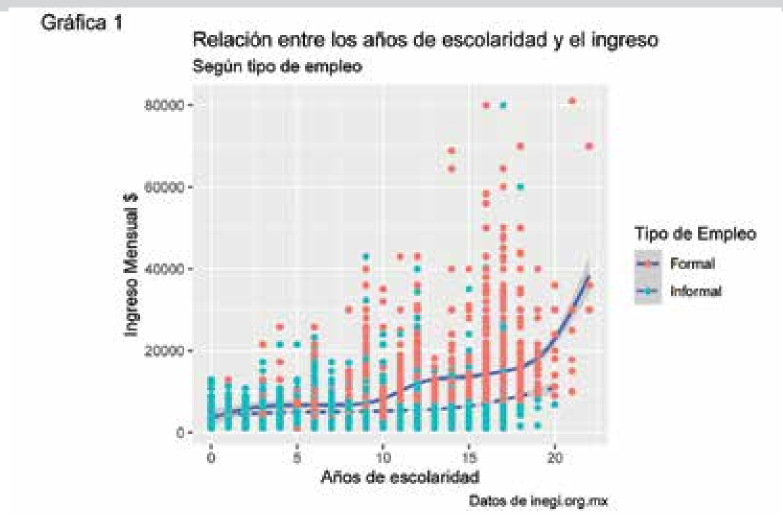
Vamos a modificar esta gráfica haciéndola un poco mas presentable. Cambiemos los nombres de los ejes, agreguemos títulos, la fuente de donde obtuvimos los datos y cambiemos el título de la leyenda.

Todas estas modificaciones que buscamos estarán contenidas en la función **labs()** la cual simplemente será un agregado al código con el cual ya contamos. Algunas de las opciones que permite este función son;

- **title** para agregar el titulo principal de la gráfica
- **subtitle** agregar subtitulo
- **caption** agregar la fuente
- **x** agregar una etiqueta al eje horizontal
- **y** agregar una etiqueta al eje vertical
- **color, linetype, size, shape** agregar el nombre de la leyenda correspondiente a la variable utilizada en el **aesthetic** la gráfica
- **tags** permite incluir texto para referenciar a la gráfica

Usando estos elementos, podemos mejorar nuestra gráfica inicial agregando estas especificaciones en **labs()**.

```
g1+
     labs(
          title = "Relación entre los años de escolaridad y el ingreso",
          subtitle = "Según tipo de empleo",
          caption = "Datos de inegi.org.mx",
          x= "Años de escolaridad",
          y= "Ingreso Mensual $",
          color="Tipo de Empleo",
          linetype="Tipo de Empleo",
          tag="Gráfica 1"
)
```



En los nombres de las etiquetas que usemos, es posible incluir ecuaciones, en caso de que esto sea necesario. Sólo debemos incluir la opción **quote**

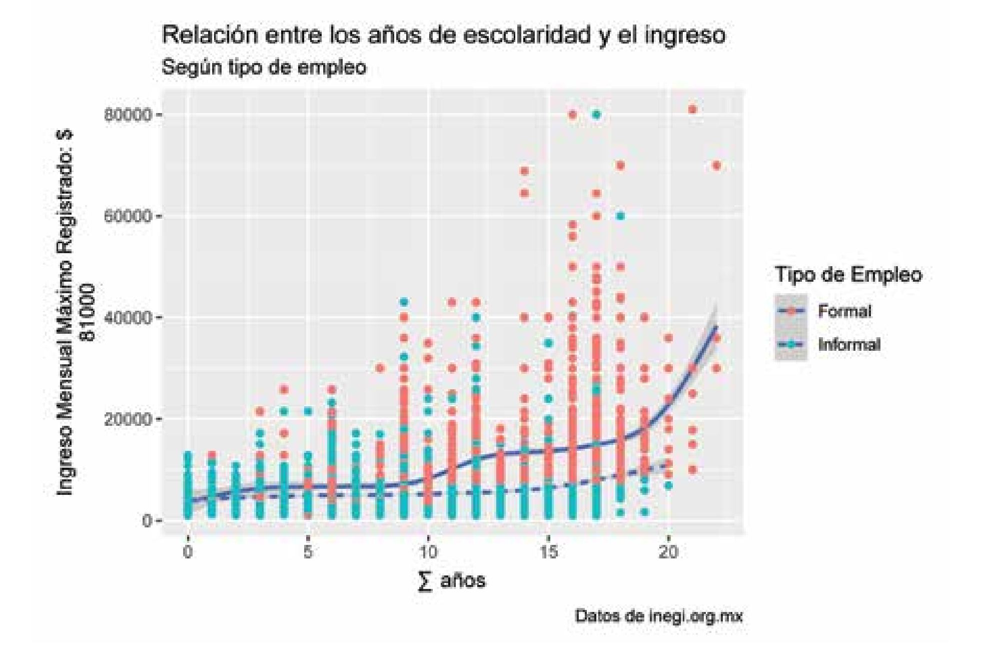
```
etiquetax <- quote(sum(años))
```

También, podríamos hacer que en la etiqueta se incluyera, por ejemplo, el valor máximo de la variable

```
etiquetay <- paste("Ingreso Mensual Máximo Registrado: $", max(enoe$ingre-
so_mensual), sep="\n")
```

De esta manera, tendríamos;

```
g1+
     labs(
          title = "Relación entre los años de escolaridad y el ingreso",
          subtitle = "Según tipo de empleo",
          caption = "Datos de inegi.org.mx",
          x= etiquetax,
          y= etiquetay,
          color="Tipo de Empleo",
          linetype="Tipo de Empleo"
)
```



4 Anotaciones

En ocasiones se vuelve necesario escribir ciertas etiquetas de texto dentro de la gráfica. Considera por ejemplo, que nos interesa determinar el tipo de empleo formal o informal de quienes tienen el máximo nivel de ingresos por cada nivel educativo. Identifiquemos primero estos niveles de ingreso.

Para ello haremos uso de las técnicas de transformación de datos repasadas con anterioridad.

```
tipo ← enoe %>%
     group_by(tipo_empleo, niv_edu, anios_esc) %>%
     summarise(max_na=max(ingreso_mensual)) %>%
     group_by(tipo_empleo, niv_edu) %>%
     filter(max_na==max(max_na))
names(tipo)[4]="ingreso_mensual"
tipo
```

```
## # A tibble: 10 x 4
## # Groups: tipo_empleo, niv_edu [8]
##    tipo_empleo niv_edu                   anios_esc ingreso_mensual
##    <chr>       <chr>                     <dbl>             <dbl>
## 1  Formal      Medio superior y superior 21                81000
## 2  Formal      Primaria incompleta       4                 25800
## 3  Formal      Prrimaria completa        8                 30000
## 4  Formal      Secundaria completa       9                 43000
## 5  Formal      Secundaria completa       11                43000
## 6  Informal    Medio superior y superior 17                80000
## 7  Informal    Primaria incompleta       4                 21500
## 8  Informal    Primaria incompleta       5                 21500
## 9  Informal    Prrimaria completa        6                 23220
## 10 Informal    Secundaria completa       9                 43000
```

El objeto de nombre **tipo** contiene los niveles de ingreso mas alto, registrado para cada uno de los nivele educativos, según el tipo de empleo. Incluimos también los años de escolaridad de la persona que reportó ese nivel de ingresos.

Definimos una gráfica *g2* en la que ya no incluyamos la linea suavizada de comportamiento

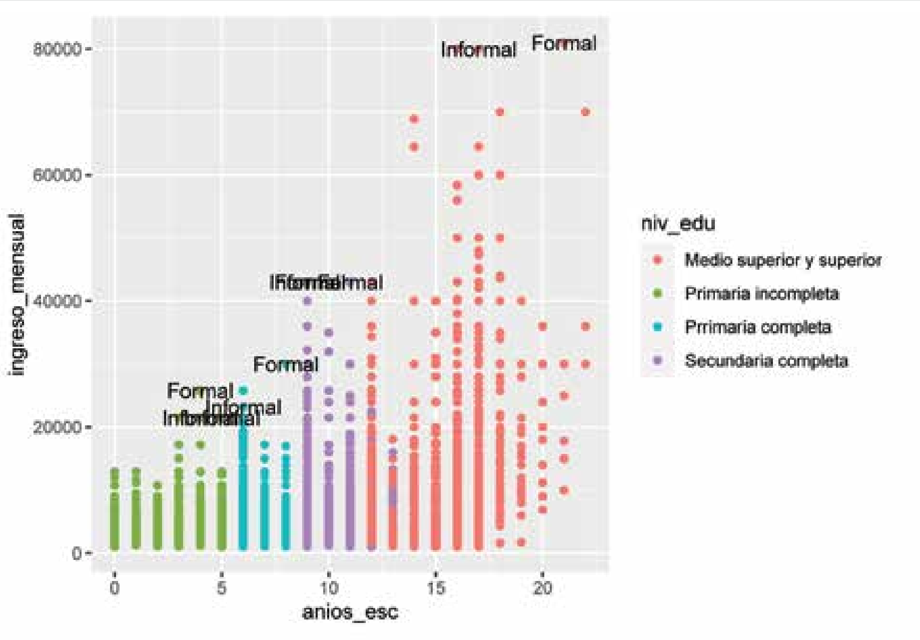
```
g2←ggplot(enoe , aes(anios_esc,ingreso_mensual))+
     geom_point(aes(color=niv_edu))
```

Tenemos dos opciones para representar nuestros datos

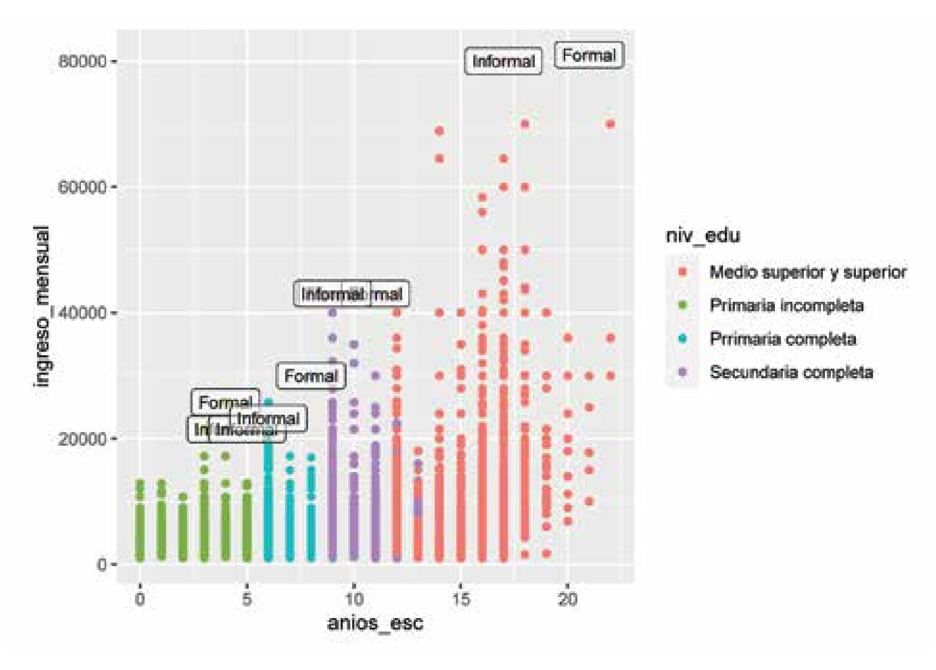
Primero con **geom_text()** pedimos que se grafiquen únicamente los puntos que están incluidos en el objeto tipo y que les agregue la etiqueta **label = tipo_empleo** a cada dato. Esto genera la siguiente gráfica.

Segundo con **geom_label** podemos incluir un pequeño recuadro en el texto.

```
g3← g2+geom_text(aes(label = tipo_empleo), data = tipo)
g3
```




```
g4← g2+geom_label(aes(label = tipo_empleo), data = tipo, alpha = 0.5, size=3)
g4
```

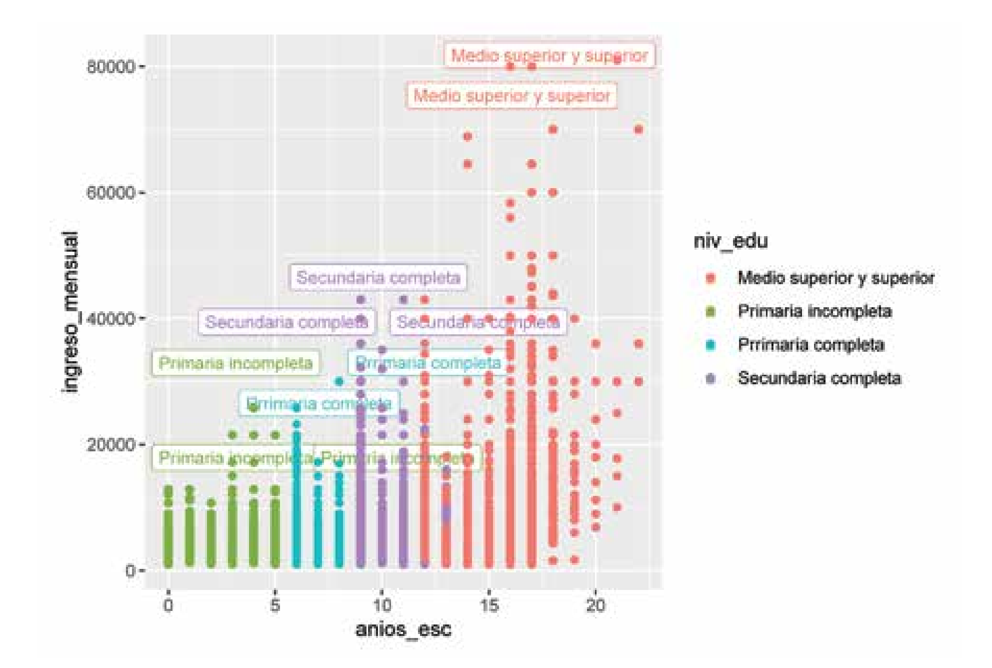


Estas gráficas son una buena representación, sin embargo, si deseamos un poco mas definición sobre los puntos, podemos usar **geom_label_repel**, función que pertenece a la librería **ggrepel**. Con estas modificaciones tenemos la siguiente representación

```
g2+
geom_point(data = tipo, alpha = 0.5, size=3)+
ggrepel::geom_label_repel(aes(label = tipo_empleo), data = tipo)
```

Con **geom_label_repel** podemos incluir una asociación entre los colores y las etiquetas. La opción **segment.color=NA** indica que en esta ocasiones no deseamos que sean visibles los pequeños segmentos que relacionan la etiqueta con el texto.

```
ggplot(enoe , aes(anios_esc,ingreso_mensual, color=niv_edu))+
     ggrepel::geom_label_repel(aes(label = niv_edu), data = tipo, size=3,
     segment.color=NA)+
geom_point()
```



Recuerda que ggplot, funciona a través de capas y que cada nueva geometría que agreguemos es una nueva capa de información. La segunda que hemos añadido ya sea con **geom_text()** o **geom_label()** incluye únicamente los puntos contenidos en el objeto de nombre **tipo**.

Considera hora que deseamos únicamente resaltar la observación que tiene el mayor nivel de ingresos. Siguiendo la lógica anterior, debemos primero identificar un objeto al que llamaremos **mayor_ing** donde se encuentre esa observación. Para ello hacemos lo siguiente;

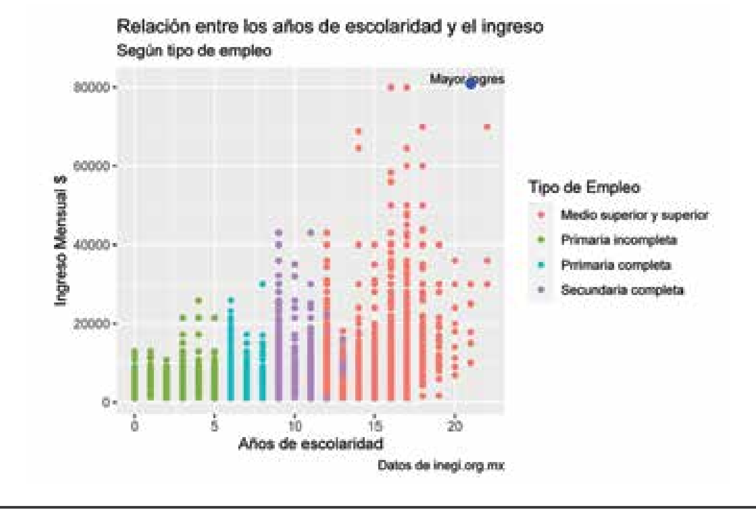

```
mayor_ing <- enoe %>%
filter(ingreso_mensual==max(ingreso_mensual)) %>%
     select(estado, niv_edu, ingreso_mensual, tipo_empleo, anios_esc) %>%
     mutate(texto="Mayor ingreso")
mayor_ing
```

```
## # A tibble: 1 x 6
##    estado         niv_edu       ingreso_mensual tipo_empleo anios_esc texto
##    <chr>          <chr>                   <dbl> <chr>           <dbl> <chr>
## 1  San Luis Po~ Medio superior y~         81000 Formal             21 Mayor in~
```

Observa que hemos incluido dentro de este objeto una columna de nombre texto con la cadena "Mayor Ingreso". Esto es para indicar que en esa posición se encuentra la etiqueta que deseamos incluir.

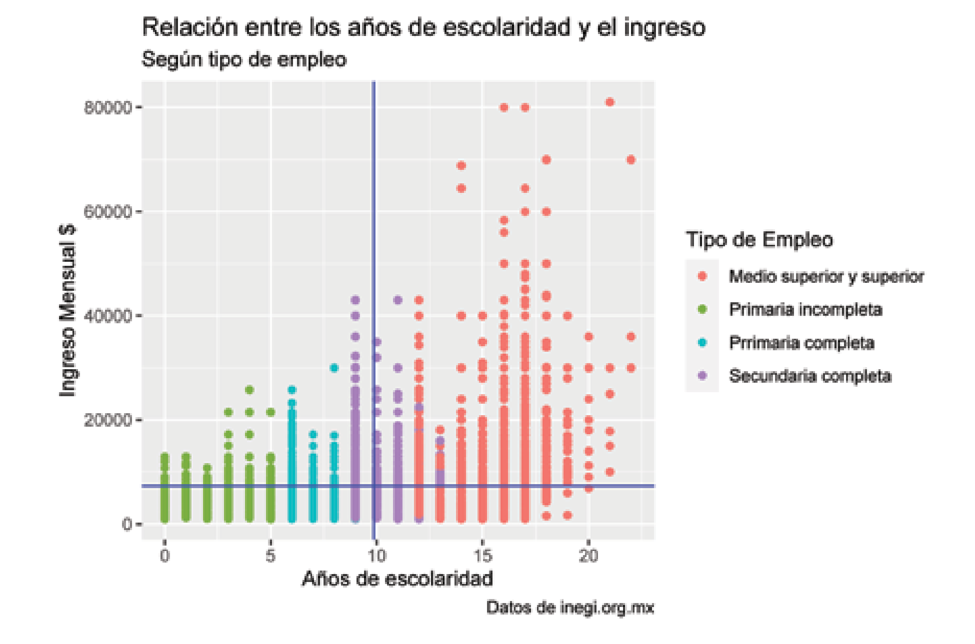
Con estos elementos efectuamos la gráfica. A la original le incluiremos dos opciones; con geom_text agregamos el punto y la etiqueta. Incluimos las opciones **hjust="center", vjust="bottom"** para indicar la posición de la etiqueta en relación con el punto donde será identificado. Volvemos a usar **geom_point** para pedirle que grafique ese único punto con un formato diferente.

```
g5 <- g2+ labs(
     title = "Relación entre los años de escolaridad y el ingreso",
     subtitle = "Según tipo de empleo",
     caption = "Datos de inegi.org.mx",
     x= "Años de escolaridad",
     y= "Ingreso Mensual $",
     color="Tipo de Empleo",
     linetype="Tipo de Empleo"
   )
g5 +
     geom_text(aes(label = texto), data = mayor_ing , size=3, color="black",
     hju t="center", vjust="bottom")+
     geom_point(aes(anios_esc,ingreso_mensual), data=mayor_ing , color="
     blue", size=3)
```



Si quisiéramos ahora agregar una línea de referencia donde se aprecie el ingreso promedio de todas las personas que contestaron la encuentra, así como los años promedio de escolaridad, podemos usar **geom_hline()** **geom_vline()**.

```
g5+geom_hline(yintercept =mean(enoe$ingreso_mensual), color="blue")+
     geom_vline(xintercept =mean(enoe$anios_esc), color="blue")
```



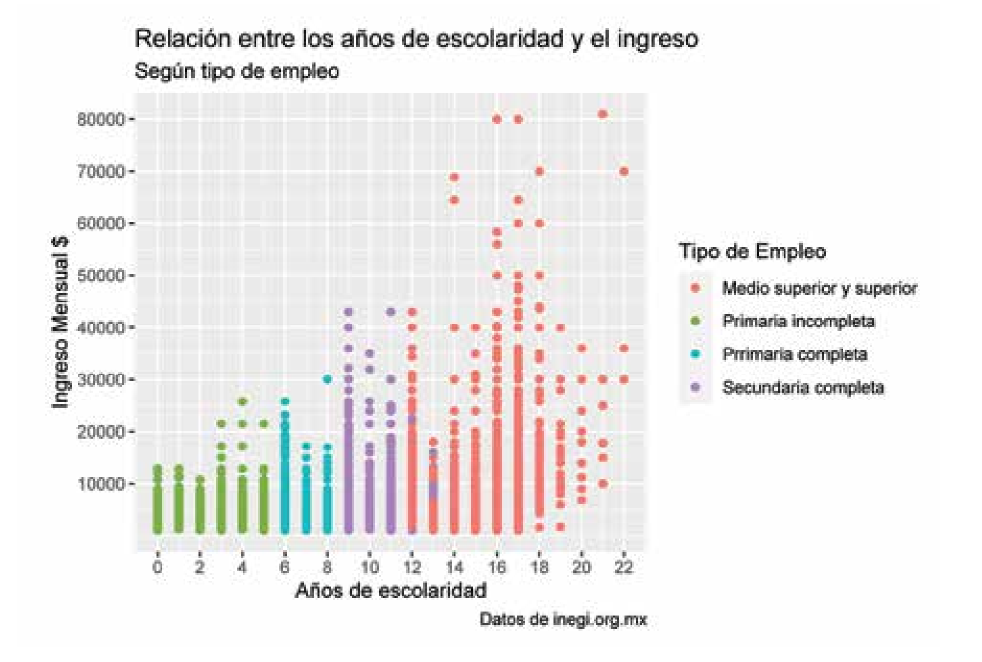
ggplot, **ggrepel** ofrecen otras mas opciones para trabajar con gráficas. Su uso sigue las mismas reglas que hemos señalado.

5 Escalas

Cuando generamos una gráfica con **ggplot2** de manera automática se ajustan los ejes de la gráfica con los valores y escalas que **ggplot** considera los adecuados. Es posible modificar estas configuraciones haciendo uso de dos funciones de esta librería, **scale_y_continuous, scale_x_continuous**. Ambas funciones cuentan con la opción **breaksy labels**. La primera indica cada cuando se mostraran las marcas divisorias de loe ejes en la gráfica, mientras que la segunda mostrara la **etiqueta** que debe acompañarla. Ambas pueden ser usadas en conjunto.

Por ejemplo, si en la gráfica anterior nos interesa cambiar las escalas para que podamos tener una mujer interpretación tenemos;

```
g5+
     scale_y_continuous(breaks = seq(10000,80000, by=10000))+
     scale_x_continuous(breaks = seq(0,26, by=2))
```

Hemos indicado que deseamos que en el eje vertical, los distintos cortes **breaks** sean comiencen en 10 mil, termines en 80 mil y tengan incrementos cada 10 mil. Algo semejante pasa con el eje x, donde incluimos una secuencia de 2 en 2. En ese caso si fuera de nuestro interés, eliminar por completo las marcas en los ejes, podríamos incluir

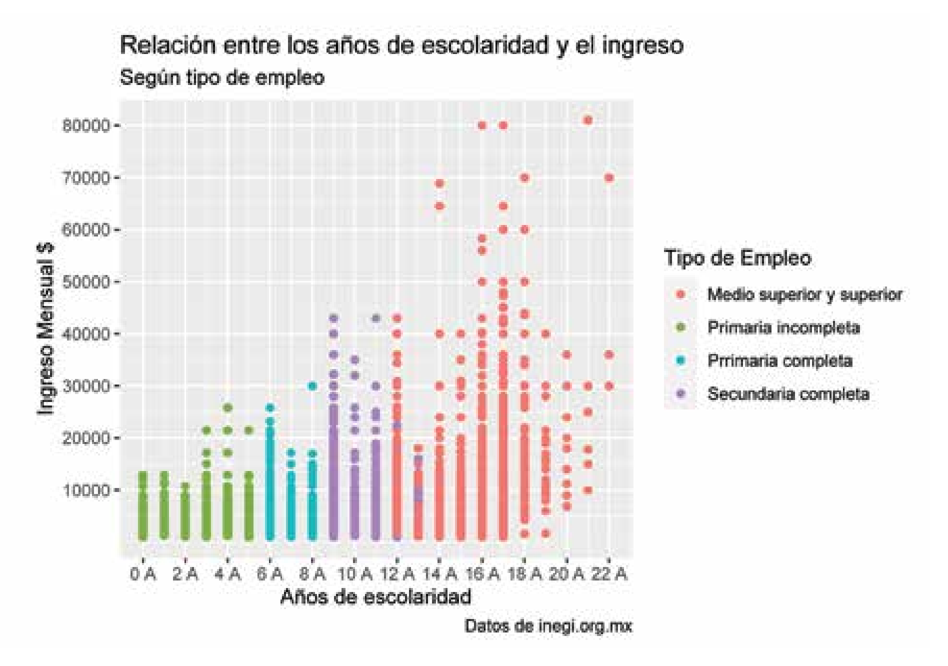
```
g5+
     scale_y_continuous(labels = NULL)+
     scale_x_continuous(labels = NULL)
```



Para usarlas de manera conjunta sólo debemos asegurarnos que el tamaño de **breaks** sea igual al tamaño de **labels**. Por ejemplo, podemos crear las siguientes etiquetas que queremos que acompañen a las marcas de los años de escolaridad.

```
et_anios <- paste(seq(0,26, by=2), "A")
```

```
g5+
     scale_y_continuous(breaks = seq(10000,80000, by=10000), labels = )+
     scale_x_continuous(breaks = seq(0,26, by=2), labels =et_anios )
```



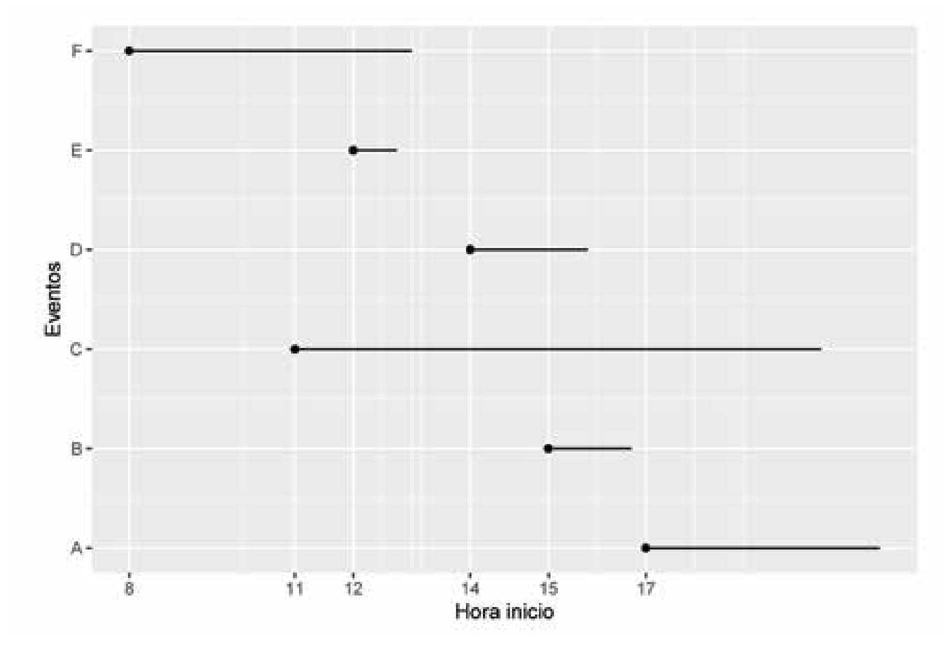
Para mostrar una utilidad mayor que puede darse a **breaks** construiremos el siguiente *data frame*, el cual suponemos tiene el inicio y el fin de seis eventos. La construcción de este *data frame* ya te s familiar.

```
inicio <- ymd_hm(paste(today(),c("17:00", "15:20", "11:00", "14:00",
"12:00","08:10")))
fin <- ymd_hm(paste(today(), c("21:00","16:45","20:00","16:00","12:45","13:00
")))
eventos <- tibble(nombre=c("A","B","C","D","E","F"),
     inicio=inicio,
     fin=fin,
     tipo=c("Religioso", "Musical", "Privado", "Musical", "Religioso", "Musi
     cal" ))
eventos
```

```
## # A tibble: 6 x 4
##    nombre     inicio                fin                   tipo
##    <chr>      <dttm>                <dttm>                <chr>
## 1  A          2021-01-31 17:00:00   2021-01-31 21:00:00   Religioso
## 2  B          2021-01-31 15:20:00   2021-01-31 16:45:00   Musical
## 3  C          2021-01-31 11:00:00   2021-01-31 20:00:00   Privado
## 4  D          2021-01-31 14:00:00   2021-01-31 16:00:00   Musical
## 5  E          2021-01-31 12:00:00   2021-01-31 12:45:00   Religioso
## 6  F          2021-01-31 08:10:00   2021-01-31 13:00:00   Musical
```


Grafiquemos la duración de estos eventos como sigue;

```
eventos %>%
     mutate(id=1+row_number())%>%
     ggplot(aes(inicio, id)) +
          geom_point() +
          geom_segment(aes(xend=fin, yend=id)) +
          scale_x_time("Hora inicio", breaks = eventos$inicio, labels =
          hour(even  tos$inicio)) +
          scale_y_continuous("Eventos", labels = eventos$nombre)
```



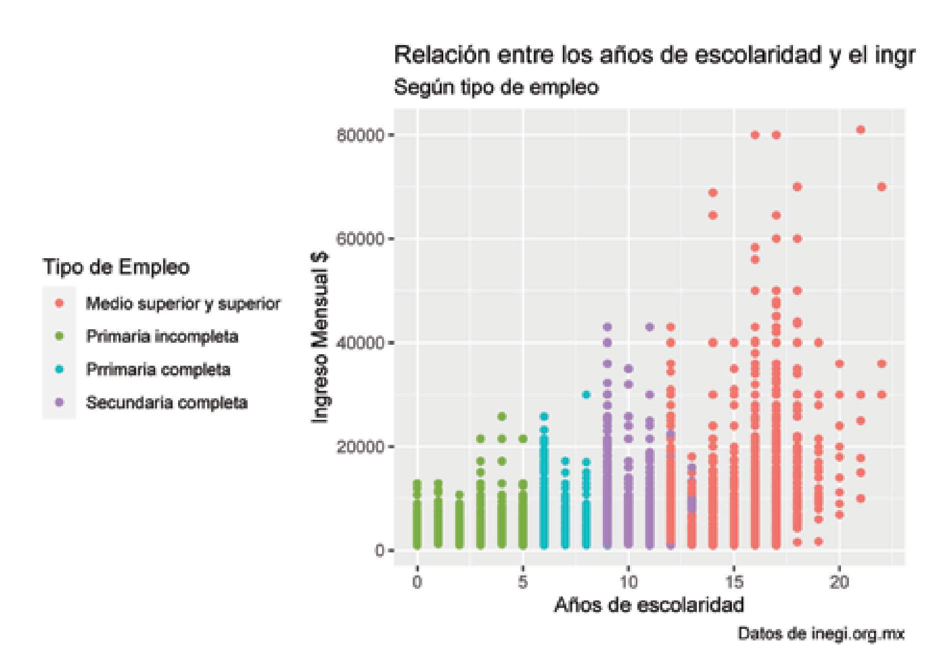
Observa que primero fue necesario otorgar un identificador numérico para cada uno de los eventos. Hemos usado **geom_segment** para indicar el fin de la línea. Esta línea empezará en cada uno de los puntos contenidos en el objeto **evento** y terminará en **fin** la cual esta contenida en el mismo objeto. En este caso hemos establecido que cada corte se efectúe al inicio del evento. También indicamos que se genere la etiqueta indicando la hora de inicio

Dentro de la variedad de manipulaciones que son posibles con el uso de **scale_** tenemos **scale_x_log10() scale_y_log10()** Estas opciones son útiles cuando se efectúa una transformación logarítmica a los datos, pues en lugar de presentar la escala en términos logarítmicos la expresa en términos continuos.

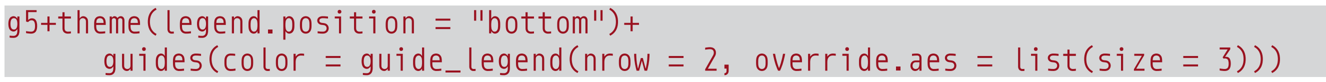
6 Leyendas

Por default *ggplot2* posiciona las leyendas de una gráfica al lado derecho. Podemos cambiar su ubicación con la función **theme(legend.position=)**

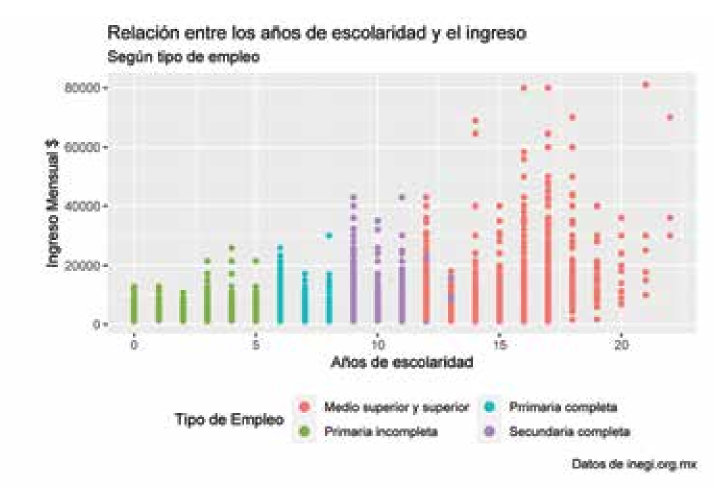
```
g5+theme(legend.position = "left")
```



Otras opciones son "top", "left", "bottom". Si usamos "none", la leyenda se eliminará. Además de decidir la ubicación de la leyenda podemos decidir su presentación con la opción **guides**. En este caso hemos indicado que se muestre la etiqueta se muestre en dos filas y que el tamaño de los puntos de la etiqueta sea 3. Todas las modificaciones que se efectúe sobre **guide_legend** tendrán efecto únicamente en la leyenda, no en los elementos de la gráfica.



```
g5+theme(legend.position = "bottom")+
     guides(color = guide_legend(nrow = 2, override.aes = list(size = 3)))
```

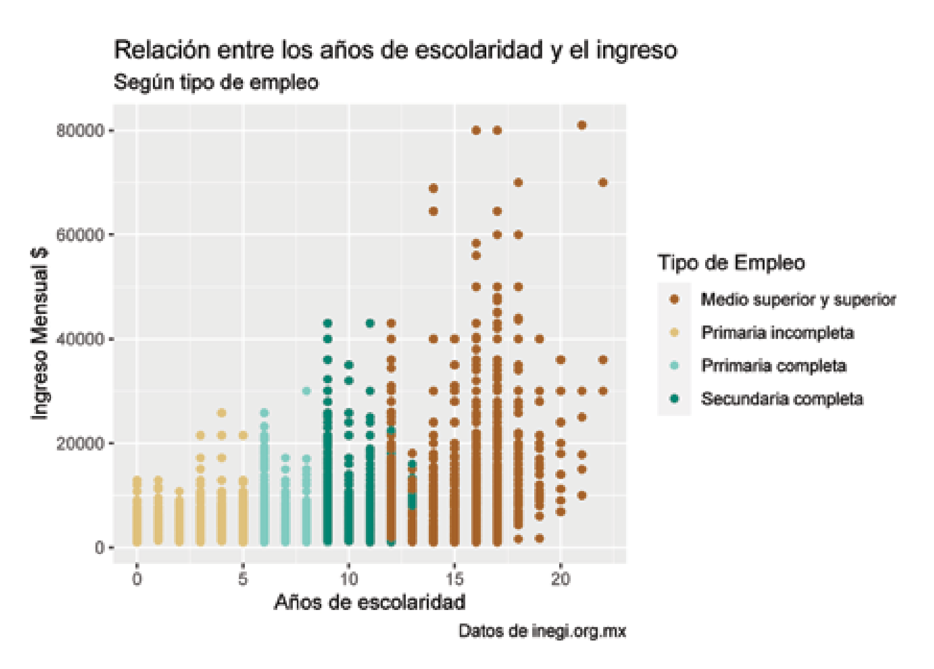


7 Colores

Anteriormente ya hemos usado una función para cambiar el color de los puntos que observamos en una gráfica, sin embargo no profundizamos mas en este tema. Para cambiar el color usamos la indicación **scale_color_brewer.**



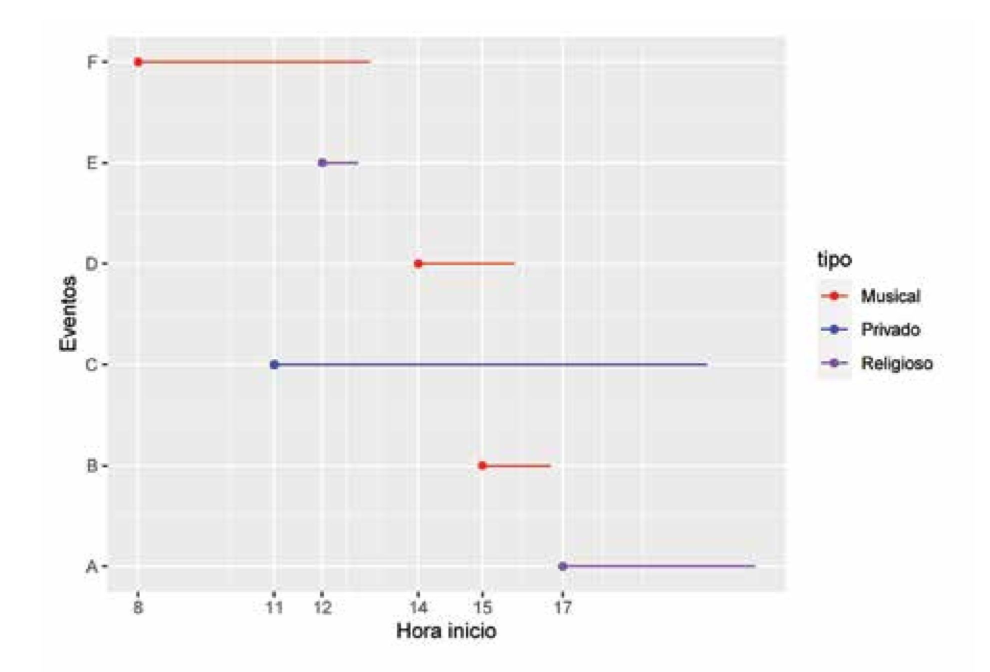
```
g5 + scale_color_brewer(palette = "BrBG")
```



Aquí puedes encontrar una lista con los nombre de las paletas de colores Colores

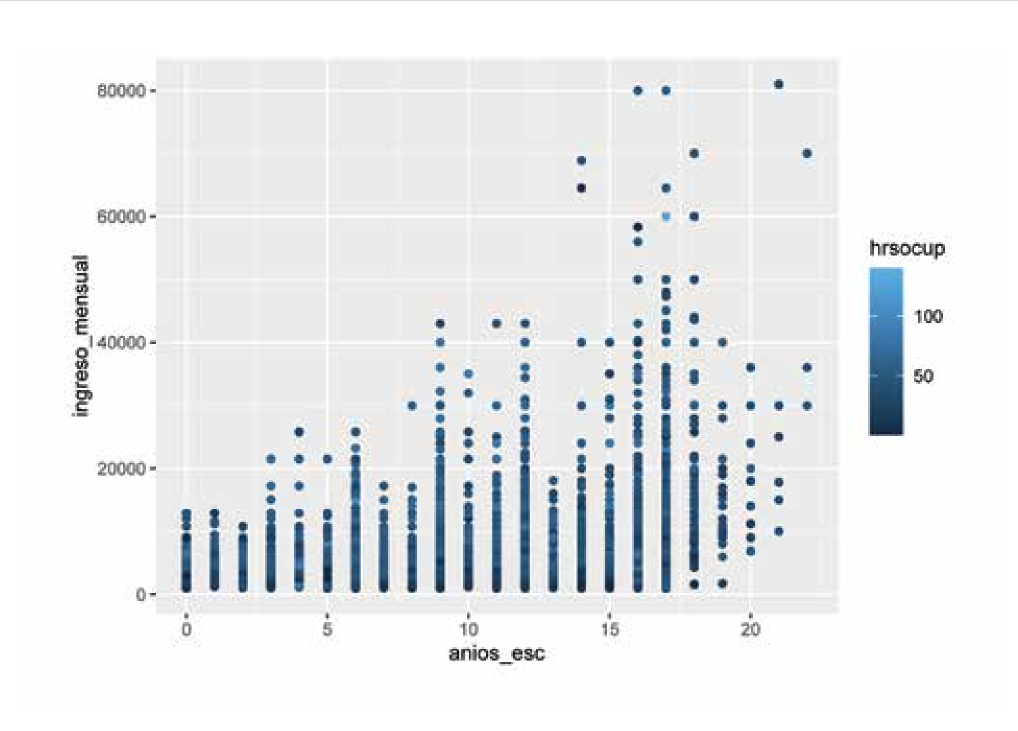
Si es de nuestro interés, seleccionar de manera manual colores, debemos usar **scale_colour_manual.** Retomando la gráfica de los eventos

```
eventos %>%
  mutate(id=1+row_number())%>%
  ggplot(aes(inicio, id, col=tipo)) +
    geom_point() +
    geom_segment(aes(xend=fin, yend=id)) +
    scale_x_time("Hora inicio", breaks = eventos$inicio, labels = hour(even
    tos$inicio)) +
    scale_y_continuous("Eventos", labels = eventos$nombre)+
    scale_color_manual(values = c(Musical="Red", Privado="Blue", Religio
    so="Purple"))
```



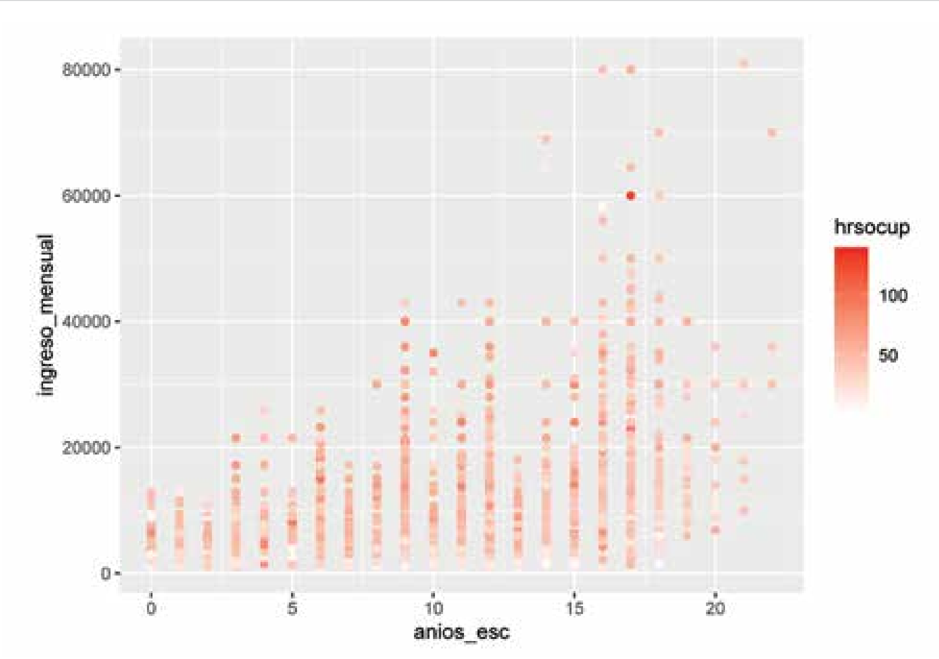
Si tenemos un conjunto de valores que son continuos y no discretos, debemos usa **scale_color_gradient()**, para indicar el color.

```
g6 <- ggplot(data=enoe)+
     geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, color=hrs
     ocup))
g6
```





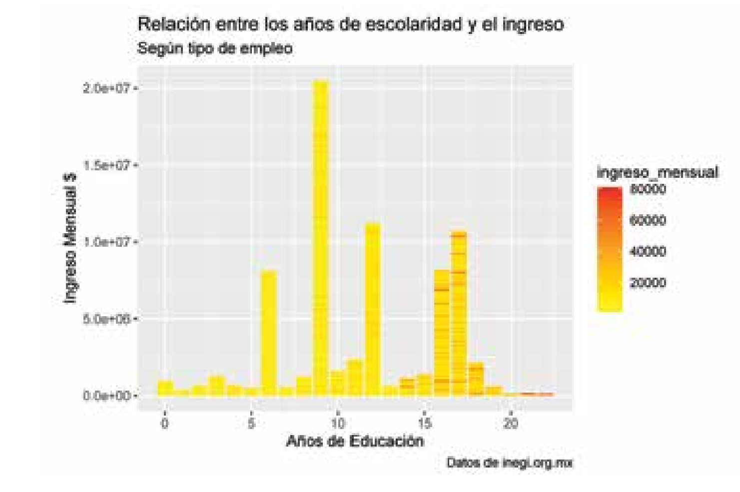
```
g6+scale_colour_gradient(low="white", high="red")
```



Si en este ejemplo los valores de la variable *hrsocup* fuesen divergentes, podríamos usar **scale_colour_gradient2**

Estas variantes para los colores que hemos visto funcionaran como aesthetics de color. Si en su lugar en la construcción de la gráfica usamos **fill** debemos cambiar color por fill.

```
ggplot(enoe, aes(x =anios_esc, ingreso_mensual )) +
   geom_bar(aes(fill = ingreso_mensual), stat = "identity") +
   scale_fill_gradient(low = "yellow", high = "red", na.value = NA)+
   labs(
      title = "Relación entre los años de escolaridad y el ingreso",
      subtitle = "Según tipo de empleo",
      caption = "Datos de inegi.org.mx",
      x= "Años de Educación",
      y= "Ingreso Mensual $"
)
```

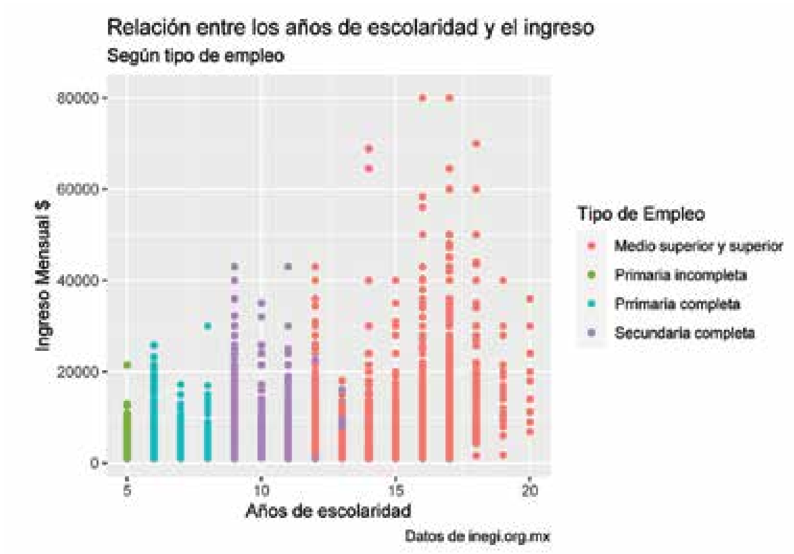


8 Manipulación de ejes

En ocasiones nos interesa únicamente una parte de los datos contenidos en una gráfica. Para recortar los ejes usamos la función **coord_cartesian** misma que permite limitar tanto el eje *x* como el *y*. Este procedimiento es equivalente a elaborar un subconjunto de los datos.



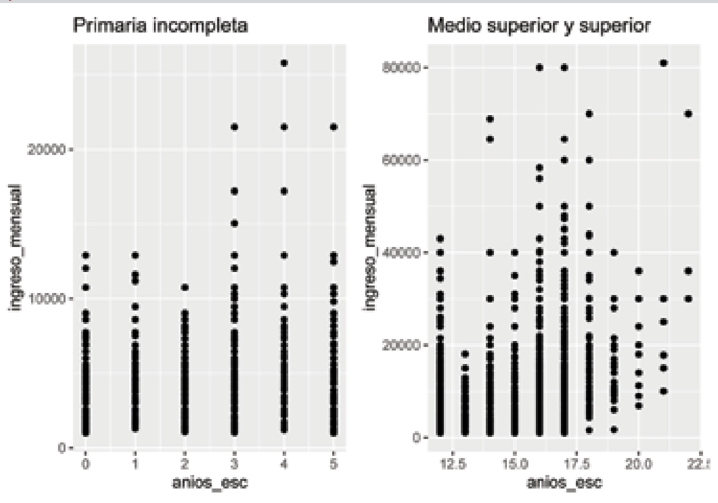
```
g5+coord_cartesian(xlim = c(5, 20))
```



Al crear una gráfica de manera automática se ajustarán los ejes para obtener la mejor visualización. En ocasiones es necesario modificar esos ejes para obtener una mejor visualización. Considera las siguientes gráficas

```
g7 <- ggplot(data=filter(enoe, niv_edu=="Primaria incompleta"))+
     geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))+
     labs(title="Primaria incompleta")

g8 <- ggplot(data=filter(enoe, niv_edu=="Medio superior y superior"))+
     geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))+
     labs(title="Medio superior y superior")
grid.arrange(g7,g8, ncol=2)
```

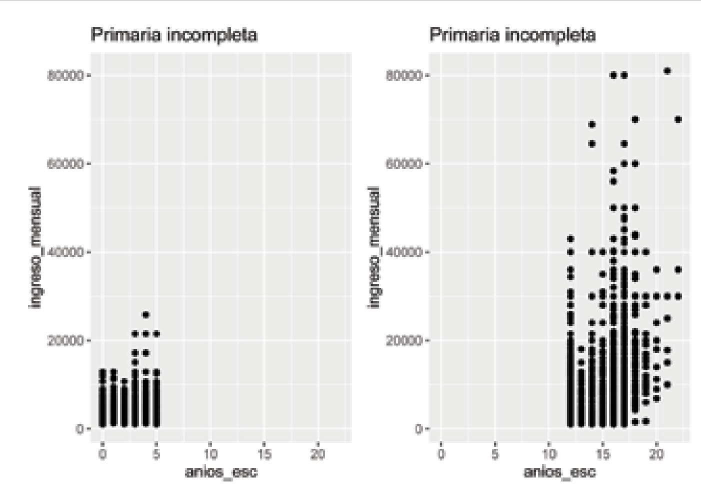


Es difícil tener una idea comparativa de los ingresos por nivel educativo, pues las escalas difieren en cada una de ellas. Para ajustarlas a la misma escala usaremos **scale_x_continuous** y **scale_y_continuous**. De esta manera hemos establecido que los limites estaran determinados por el rango de cada una de las variables que incluimos en la gráfica. Usaremos estas escalas para modificar ambas graficas y ajustar a la escala propuesta

```
escalax <- scale_x_continuous(limits = range(enoe$anios_esc))
escalay <- scale_y_continuous(limits = range(enoe$ingreso_mensual))
```

```
g7 <- ggplot(filter(enoe, niv_edu=="Primaria incompleta"), aes(anios_esc, in-
greso_mensual))+
     geom_point()+
     labs(title="Primaria incompleta") +
     escalax +
     escalay

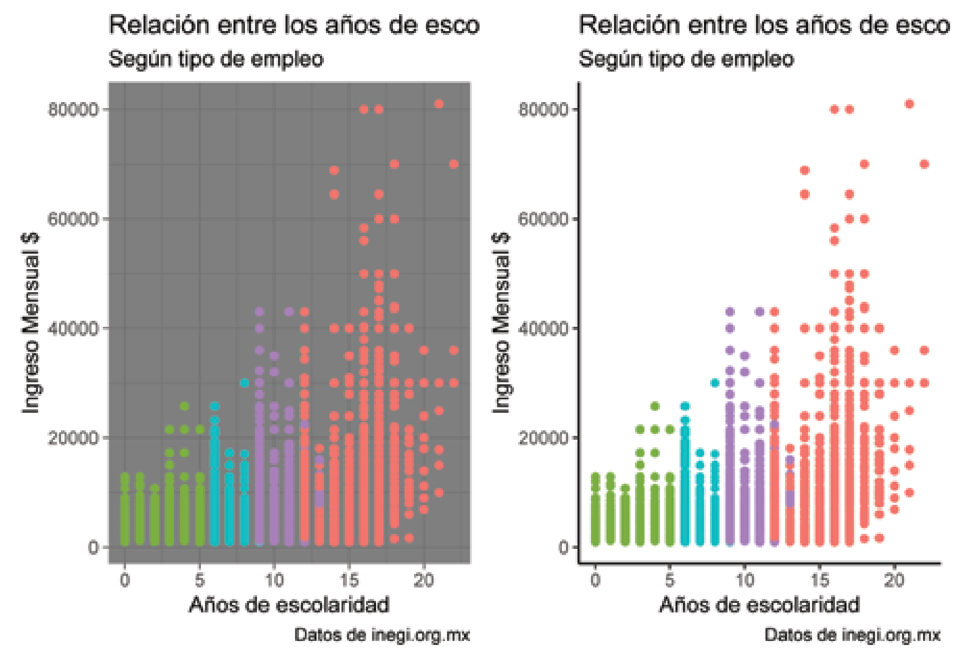
g8 <- ggplot(filter(enoe, niv_edu=="Medio superior y superior"), aes(anios_
esc, ingreso_mensual))+
     geom_point()+
     labs(title="Primaria incompleta") +
     escalax +
     escalay
grid.arrange(g7,g8, ncol=2)
```



9 Temas

El último de los elementos que vamos a modificar en nuestra gráficas es el tema. *ggplot2* cuenta 8 temas que nos permiten modificar el entorno de la gráfica. Por ejemplo los temas **theme_bw()**

```
g9 <- g5+theme_dark()+theme(legend.position = "NULL")
g10 <- g5+theme_classic()+theme(legend.position = "NULL")
grid.arrange(g9,g10, ncol=2)
```



El resto de temas con los que puedes probar son;

- `theme_grey default`
- `theme_gray`
- `theme_bw`
- `theme_linedraw`
- `theme_light`

10 Tamaño textos y colores

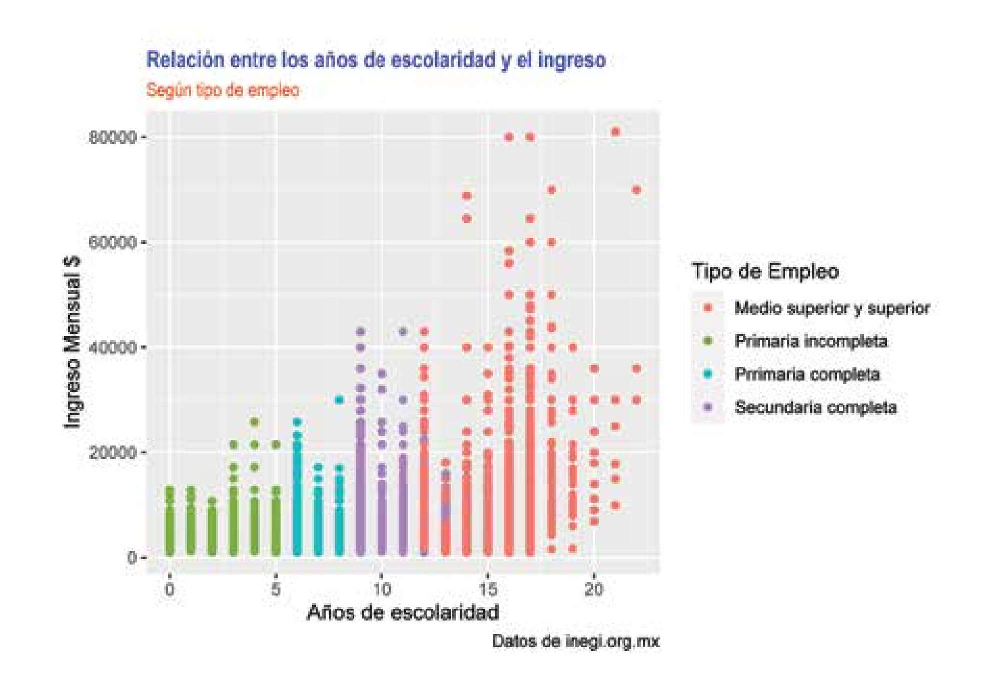
En los ejemplos anteriores usamos la función **theme** para ajustar la posición de la leyenda. Esta función tiene muchas mas opciones que en combinación con **element_text** permiten modificar la apariencia de los diferentes elementos de la gráfica.

Consideremos que deseamos modificar el texto del titulo y subtitulo de una gráfica. Definamos primero que el formato que deseamos. Para ello usamos **element_text**

```
titulo_grafica ← element_text(family = "Arial Narrow", face = "bold", co-
lor="blue", size=12)
subtitulo_grafica← element_text(family = "Arial Narrow", color="red",
size=10)
```

Una vez definidos estos elementos, solo los aplicamos a la gráfica usando **theme**.

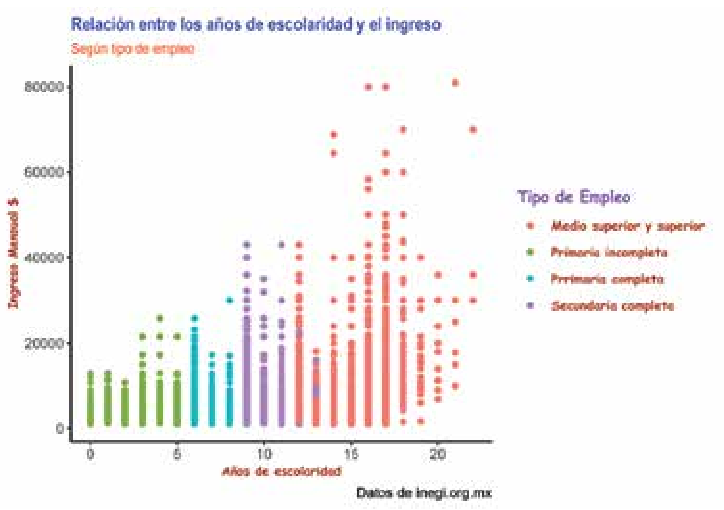
```
g5+ theme(plot.title = titulo_grafica, plot.subtitle =subtitulo_grafica)
```

Con la función **theme** y **element_text** podemos modificar cualquier elemento de texto contenido en la gráfica

```
texto_ejes← element_text(family = "Comic Sans MS", face = "bold", color="-
brown", size=8)
texto_ley ← element_text(family = "Comic Sans MS", face = "bold", co-
lor="purple", size=10)
```

```
g5+theme_classic()+
     theme(plot.title = titulo_grafica,
          plot.subtitle =subtitulo_grafica,
          axis.title.x = texto_ejes,
          axis.title.y = texto_ejes,
          legend.text=texto_ejes,
          legend.title= texto_ley)
```



Si estamos trabajando con múltiples gráficos y deseamos ser consistentes en ellos, podemos definir un objeto que contenga todos los elementos y simplemente sobreponerlos a las distintas gráficas.

```
mitema ←theme_classic()+ theme(plot.title = titulo_grafica,
     plot.subtitle =subtitulo_grafica,
     axis.title.x = texto_ejes,
     axis.title.y = texto_ejes,
     legend.text=texto_ejes,
     legend.title= texto_ley)
```

```
ggplot(data=enoe)+
     geom_bar(mapping = aes(x=sex, fill=niv_edu))+
     mitema
```

Cuanto trabajamos con archivos tipo *Rmarkdown* puede ser complicado cambiar el tipo de letra, ya que en algunos formatos de salida, por ejemplo pdf, esto puede generar errores. En este caso lo mas recomendable es instalar las librerías e importar los diferentes tipos de fuente. Una manera de hacerlo es usando;

```
library(extrafont)
extrafont::loadfonts(quiet=TRUE)
extrafont::font_import()
```

Ejecutar esta instrucción tomará algún tiempo a la computadora.

11 Guardar

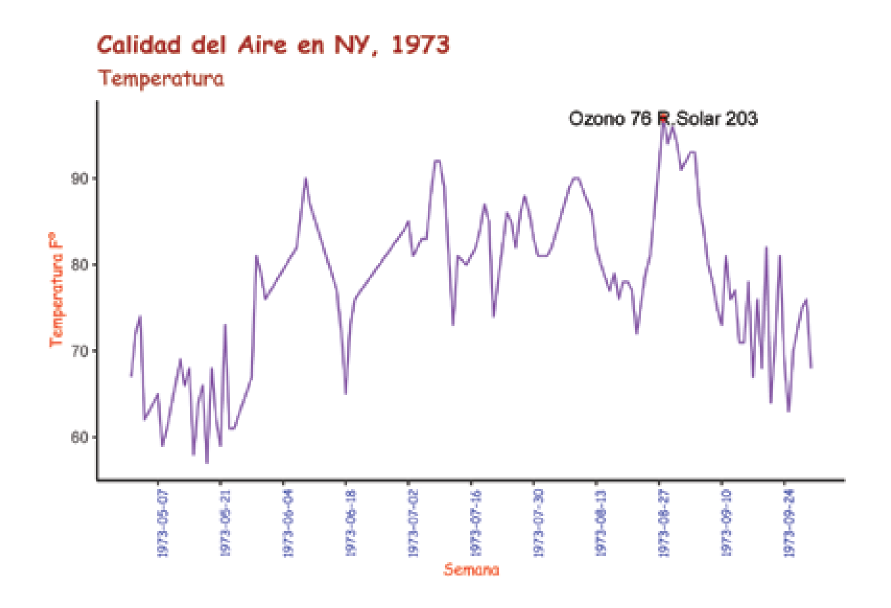
Para guardar una gráfica efectuada es necesario usar el comando **ggsave**, debemos incluir el nombre que daremos a la gráfica acompañado de comillas, incluyendo la indicación sobre la extensión del formato. Debemos también indicar el nombre de la gráfica que deseamos guardar. El archivo se guardará en el directorio de trabajo que hayamos definido. Si no asignamos el nombre de la gráfica que deseamos guardar, de manera automática se almacenará la última gráfica elaborada.

```
ggsave("Gráfica_5.pdf", g5)
```

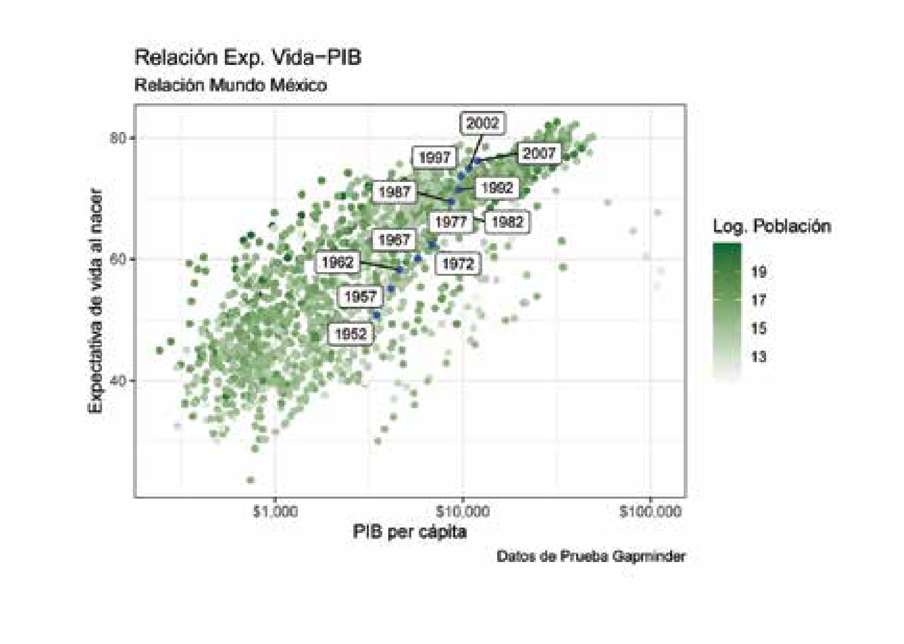
```
## Saving 6.5 x 4.5 in image
```

12 Actividades

1. La base de datos *airquality* (dentro de la librería datasets) contiene los datos registrados de la calidad del aire en Nueva York del mes de Mayo a Septiembre de 1973. Esta base forma parte de los data set de ejemplo de **R**. Úsala para construir la siguiente gráfica. Para este ejercicio deberás revisar la documentación de **theme** y hacer uso de varios de los elementos que aprendimos en capítulos anteriores. Para los cortes del eje horizontal usa la función **scale_x_date(breaks="2weeks")**



2. Usa la base de datos **gapminder** (dentro de la librería del mismo nombre) que contiene información sobre diferentes variable macroeconomicas de varios países del mundo. En estos países se encuentra México. Usa la información para replicar la siguiente gráfica. La población está expresada en términos logarítmicos **log(pop)** y las etiquetes del eje "x" las obtuvimos usando **scale_x_log10(labels = scales::dollar)**. Los puntos que sobresalen al resto corresponden a la evolución de México.



Apéndice A

Soluciones al Capítulo 2 | Básicos para el uso de R

Contents

1. Genera los objetos necesarios para que z sea el resultado del área de un círculo.

```
radio <- 5
z <- (pi*radio^2)
```

2. Sí el radio del es 5, ¿cuál es el valor del área del circulo?

```
z
```

```
## [1] 78.53982
```

3. Genera una secuencia de números que tomen valores de 0.5 en 0.5 comenzando en 1 y terminando en 100

```
seq(1,100, by=0.5)
```

```
## [1]     1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5
## [13]    7.0  7.5  8.0  8.5  9.0  9.5 10.0 10.5 11.0 11.5 12.0 12.5
## [25]   13.0 13.5 14.0 14.5 15.0 15.5 16.0 16.5 17.0 17.5 18.0 18.5
## [37]   19.0 19.5 20.0 20.5 21.0 21.5 22.0 22.5 23.0 23.5 24.0 24.5
## [49]   25.0 25.5 26.0 26.5 27.0 27.5 28.0 28.5 29.0 29.5 30.0 30.5
```

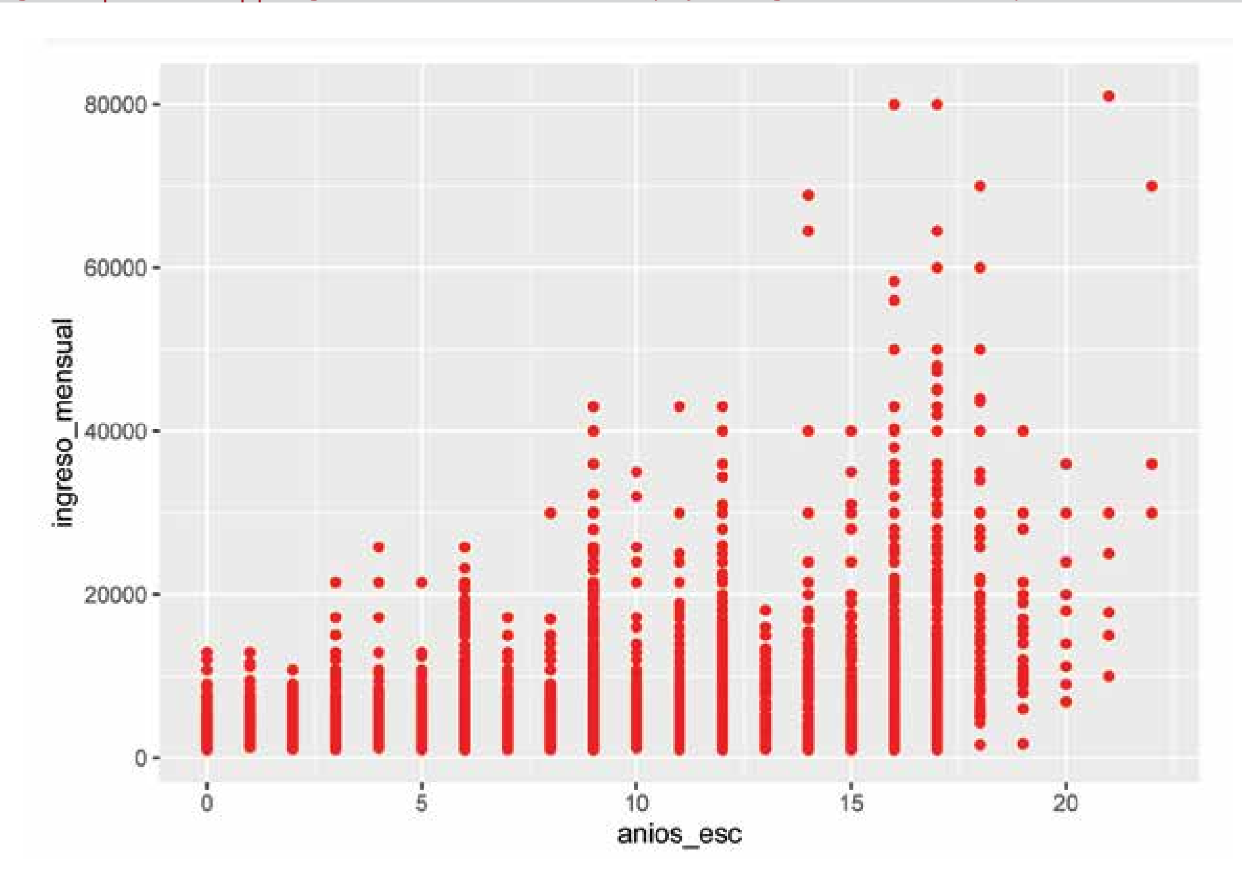


```
## [61]    31.0 31.5 32.0 32.5 33.0 33.5 34.0 34.5 35.0 35.5 36.0 36.5
## [73]    37.0 37.5 38.0 38.5 39.0 39.5 40.0 40.5 41.0 41.5 42.0 42.5
## [85]    43.0 43.5 44.0 44.5 45.0 45.5 46.0 46.5 47.0 47.5 48.0 48.5
## [97]    49.0 49.5 50.0 50.5 51.0 51.5 52.0 52.5 53.0 53.5 54.0 54.5
## [109]   55.0 55.5 56.0 56.5 57.0 57.5 58.0 58.5 59.0 59.5 60.0 60.5
## [121] 61.0 61.5 62.0 62.5 63.0 63.5 64.0 64.5 65.0 65.5 66.0 66.5
## [133] 67.0 67.5 68.0 68.5 69.0 69.5 70.0 70.5 71.0 71.5 72.0 72.5
## [145] 73.0 73.5 74.0 74.5 75.0 75.5 76.0 76.5 77.0 77.5 78.0 78.5
## [157] 79.0 79.5 80.0 80.5 81.0 81.5 82.0 82.5 83.0 83.5 84.0 84.5
## [169] 85.0 85.5 86.0 86.5 87.0 87.5 88.0 88.5 89.0 89.5 90.0 90.5
## [181] 91.0 91.5 92.0 92.5 93.0 93.5 94.0 94.5 95.0 95.5 96.0 96.5
## [193] 97.0 97.5 98.0 98.5 99.0 99.5 100.0
```

Soluciones al Capítulo 3 | Visualización con ggplot2

1. Elabora una gráfica de dispersión que permita observar la relación entre el salario mensual y la edad. Donde todos los puntos sean rojos

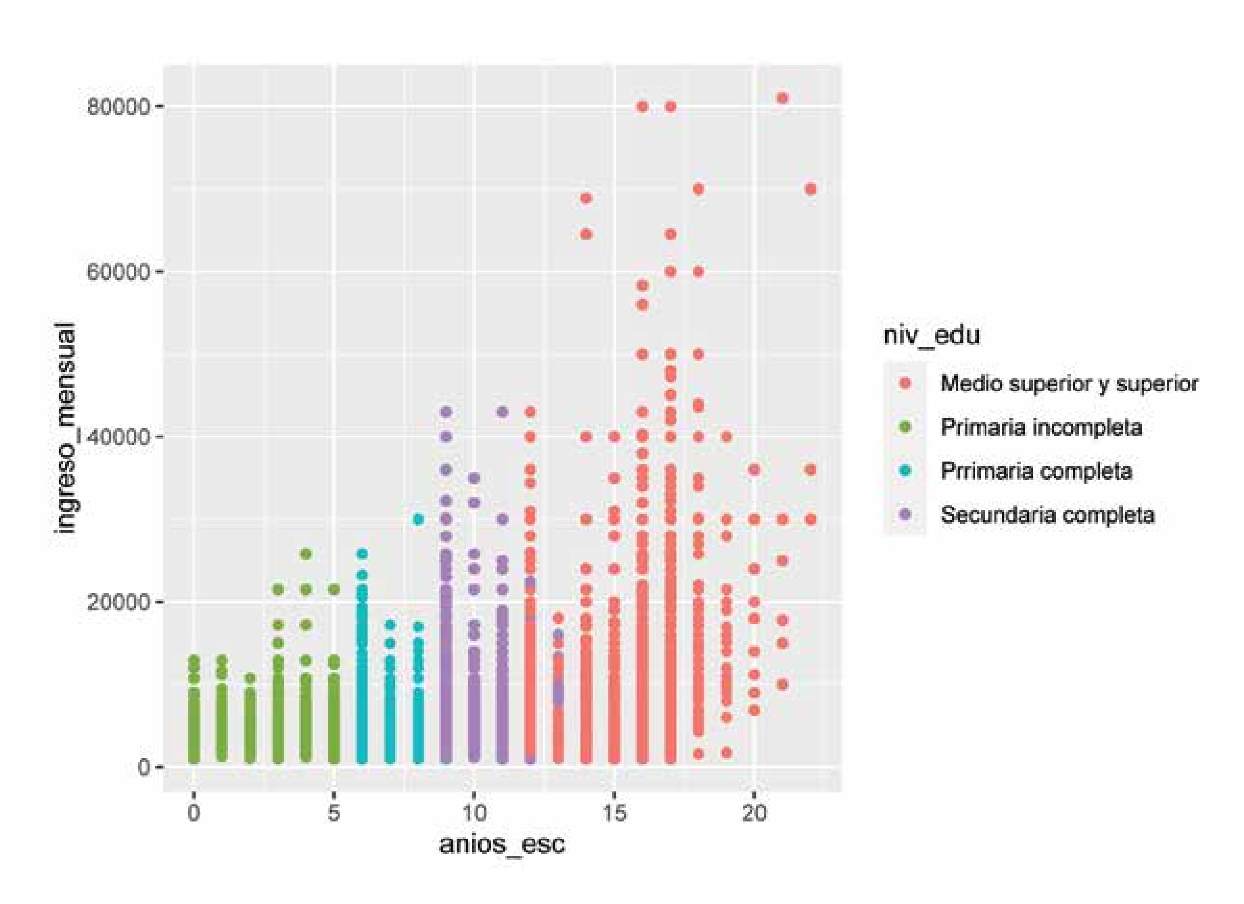
```
ggplot(data = enoe) +
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual), color="red")
```



En este caso la indicación de color debe ir fuera de las opciones de **aes**, ya que deseamos que todos los puntos sean de color rojo.

2. Elabora una gráfica que permita observar la relación entre el ingreso entre el salario mensual y la edad. Además que los puntos permitan identificar el nivel de escolaridad.

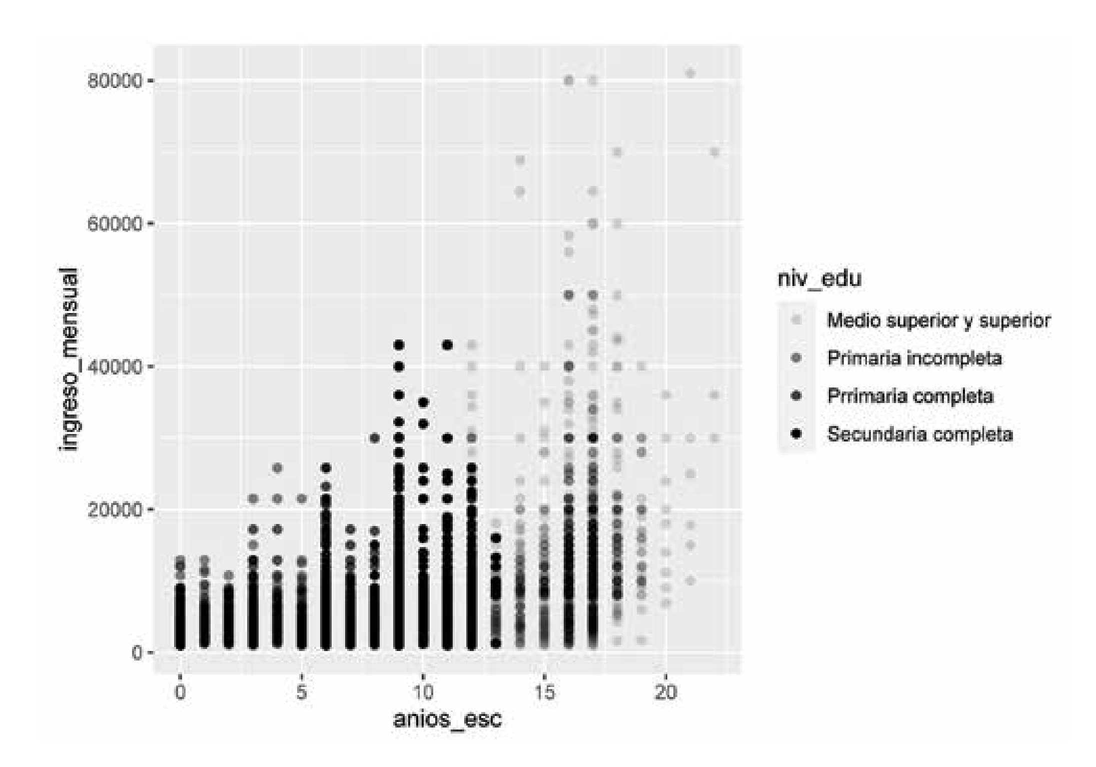
```
ggplot(data = enoe) +
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, color=niv_edu))
```



En este caso la indicación sobre el color debe incluirse dentro de la opción aes.

3. Intenta efectuar una gráfica de puntos, donde en el eje x se observen los años de educación, en el eje y, el ingreso mensual y ademas usando la estética de alpha sobre la variable nivel de educación. ¿Cuál es el warning que emite R? ¿Porqué crees que sucede esto?

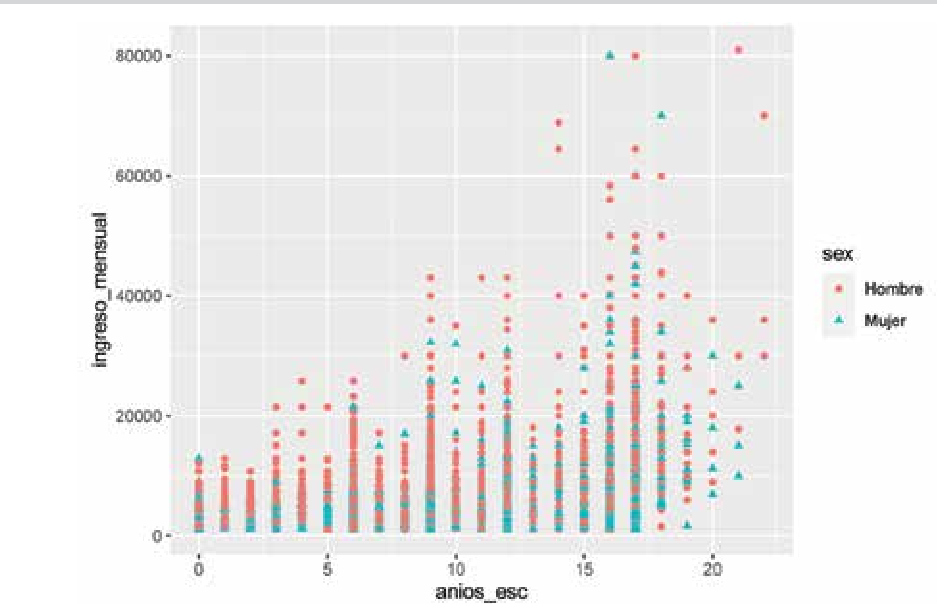
```
ggplot(data = enoe) +
  geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, alpha=niv_edu ))
## Warning: Using alpha for a discrete variable is not advised.
```



En este caso obtenemos el warning **Using alpha for a discrete variable is not advised**. Recuerda que tanto **alpha** como **size** se utilizan sobre variables de tipo numerico y la variable sobre el nivel de educación (niv_edu) es de tipo categorico.

4. Elabora una gráfica que permita ver la relación del ingreso mensual con los años de educación diferenciada por sexo, de manera que cambie no sólo el color del punto, sino ademas la forma

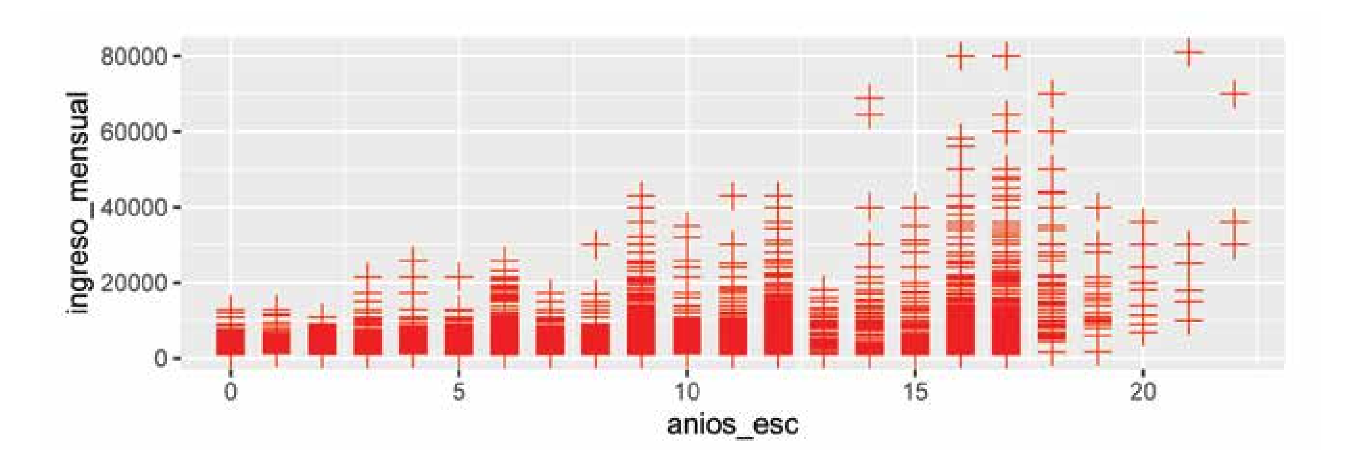
```
ggplot(data = enoe) +
     geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, shape=sex,
     color=sex ))
```

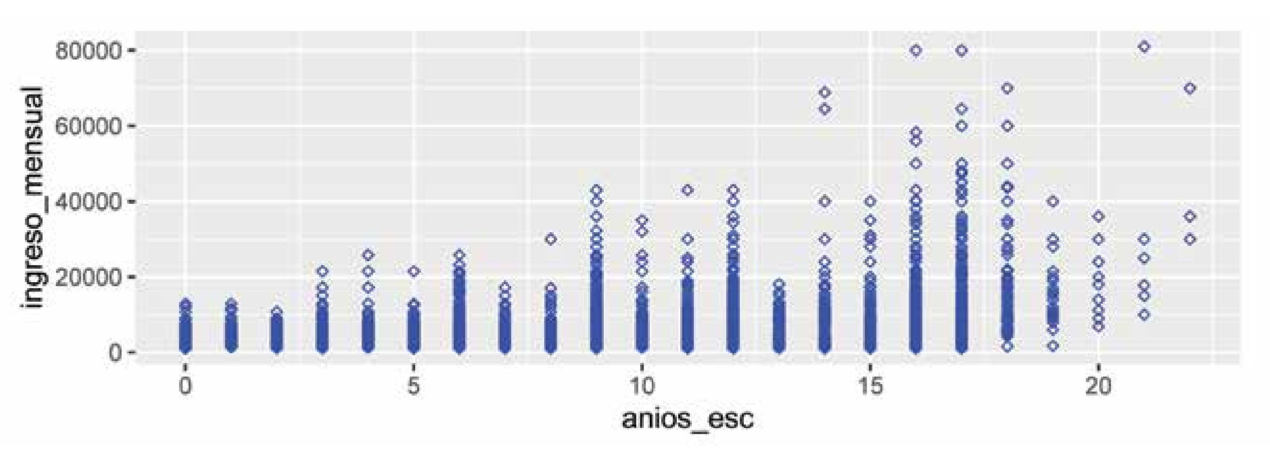


En este caso dentro de la **aes** debemos incluir tanto la opción **shape** como **color**, debido a que diferenciamos por sexo, en los dos casos indicamos la mismo variable.

5. Construye las siguientes gráficas, analiza detenidamente las diferencias entre ellas y responde de las preguntas.

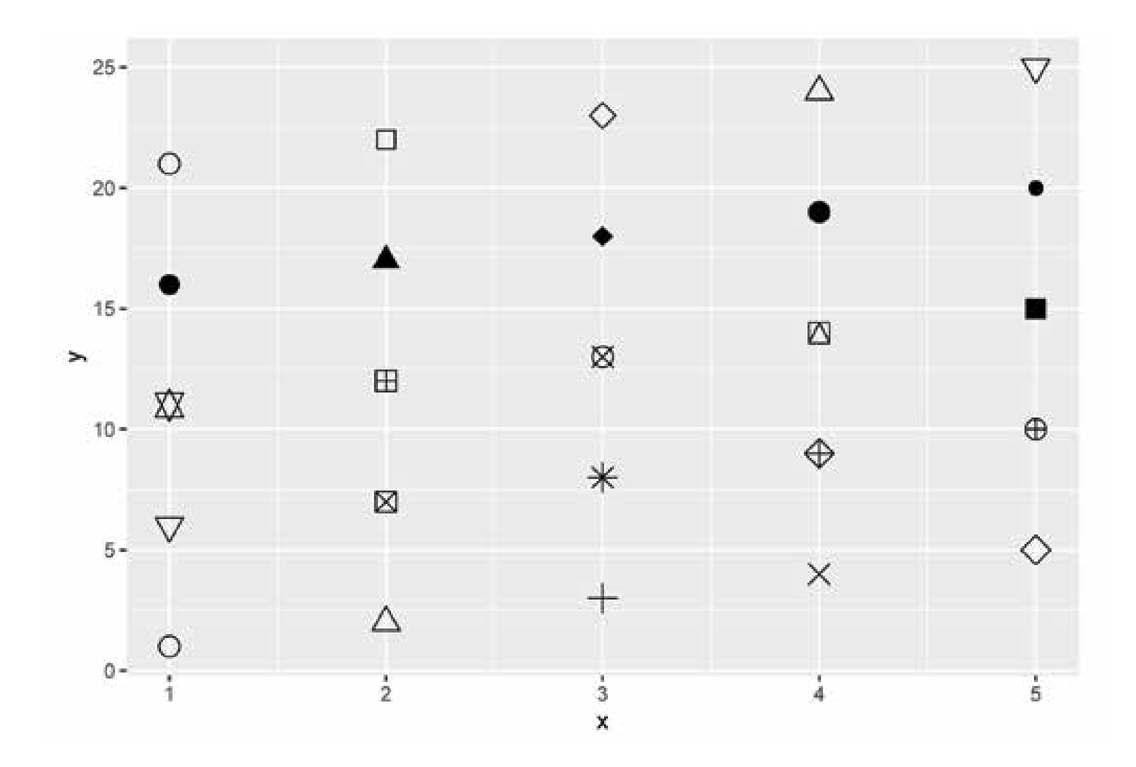
- `ggplot(data = enoe) + geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual), shape=3, size=3, color="red")`
- `ggplot(data = enoe) + geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual), shape=5, size=1, color="blue")`





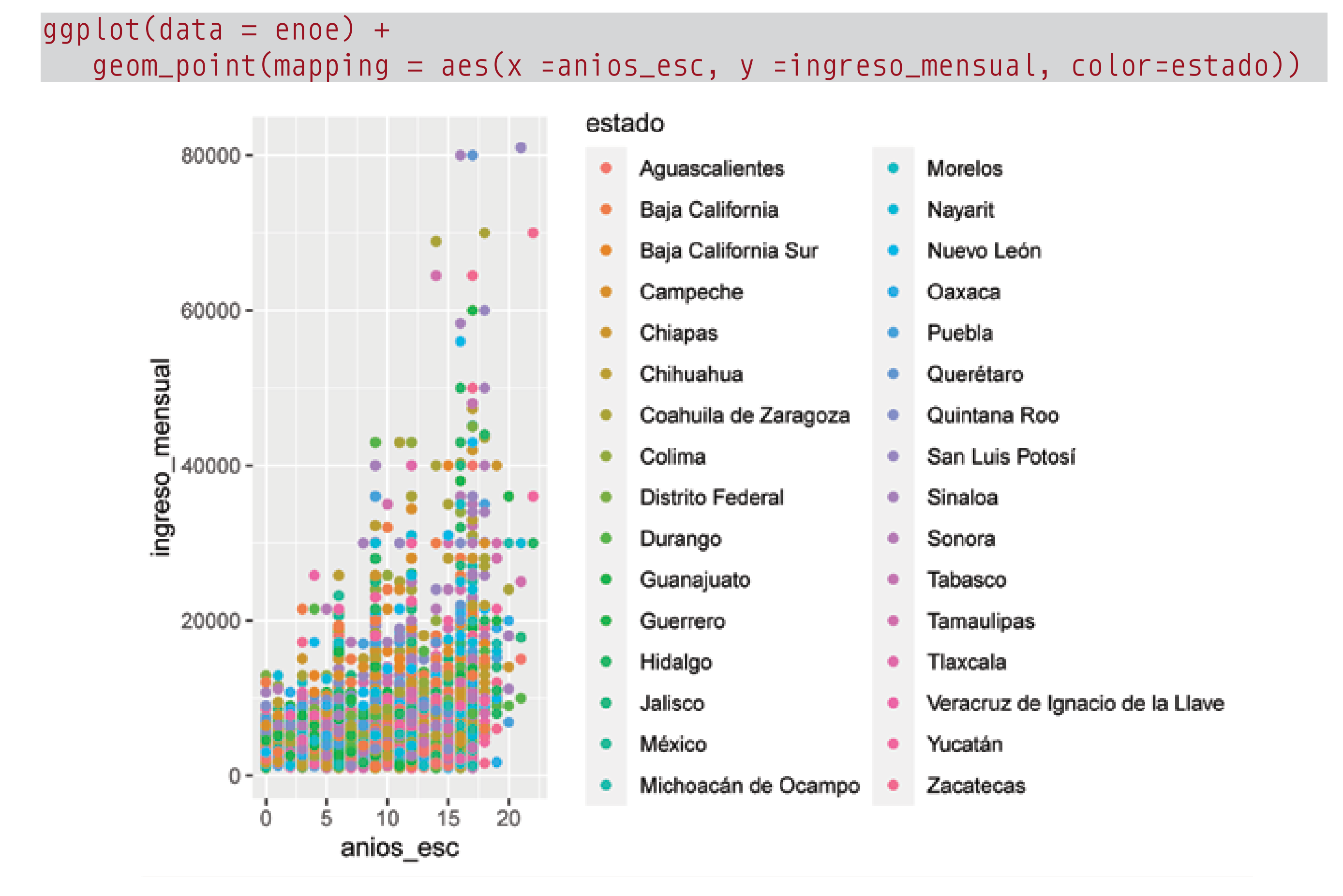
- ¿Qué impacto tiene sobre la gráfica la opción shape, size y color cuando éstas se incluyen fuera de la función **aes**?

Al incluir estas funciones fuera de aes, se modifican todos los puntos; shape permite el cambio de forma, size el tamaño y color el color observado. Con **ggplot2** podemos elegir entre diferentes formas para representar puntos. Cada forma esta representada por un número, la siguiente tabla



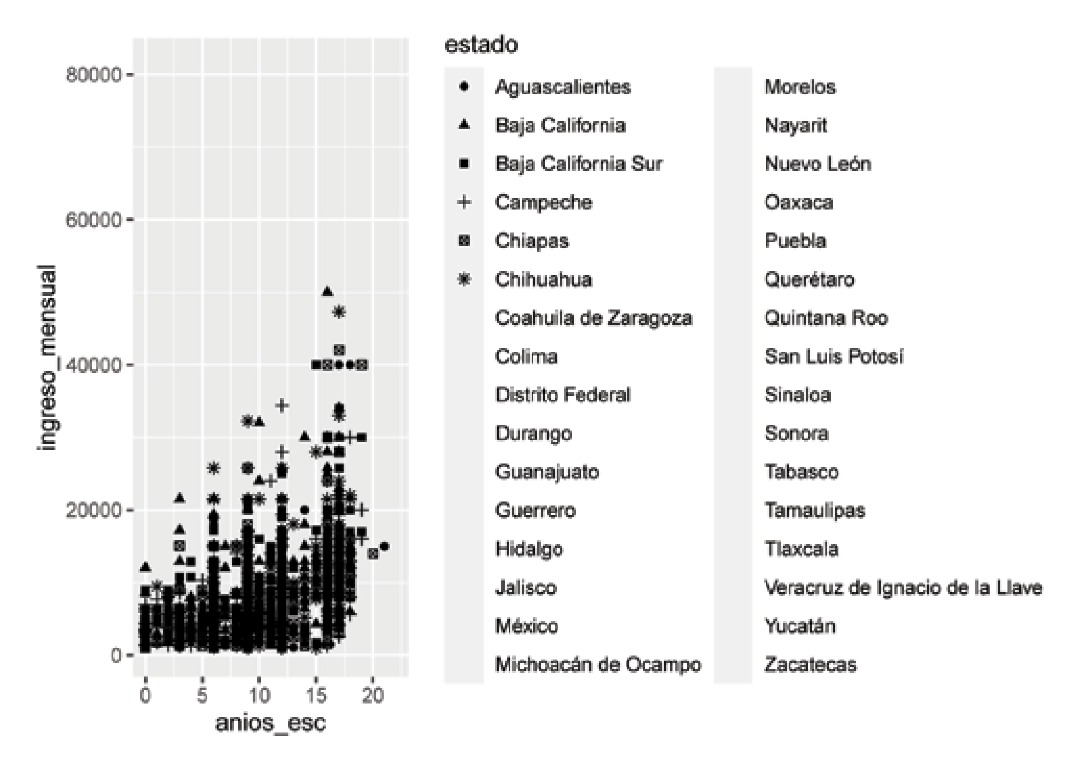
6. Elabora una gráfica que permita observar la relación la edad y el ingreso mensual, diferenciando por estados usando primero color y después shape. ¿Observas alguna inconsistencia?

```
ggplot(data = enoe) +
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, color=estado))
```




```
ggplot(data = enoe) +
   geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual, shape=estado))
```

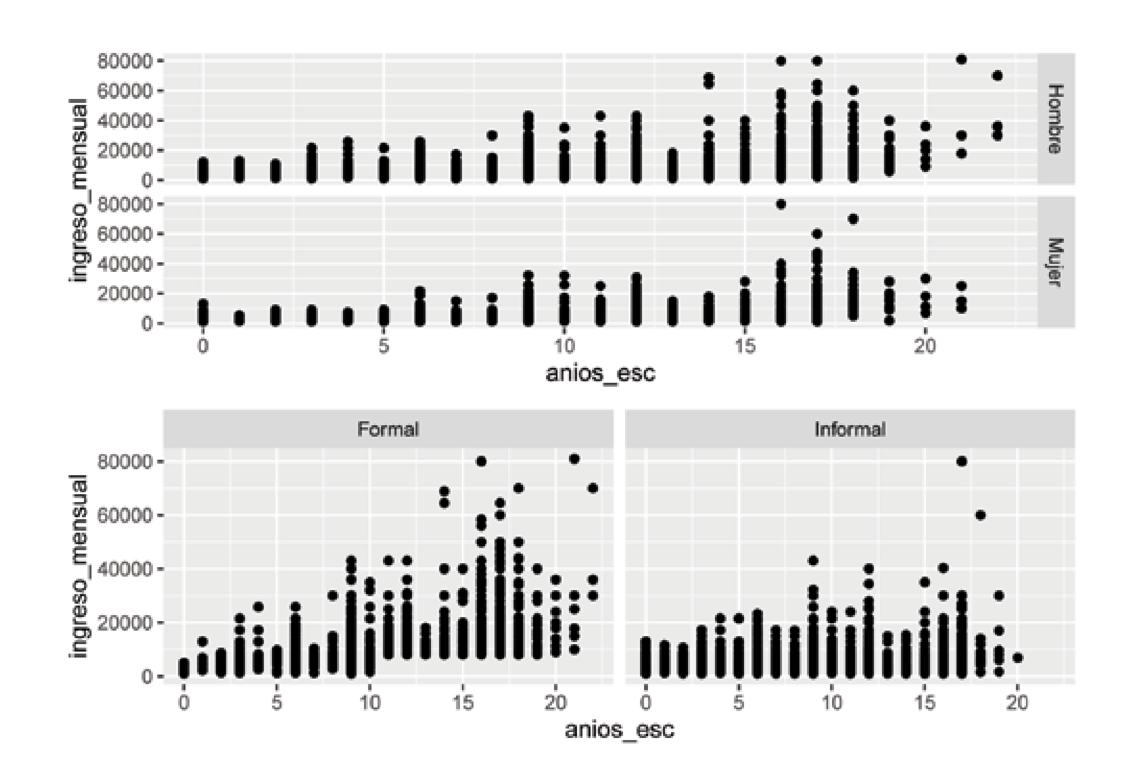
```
## Warning: The shape palette can deal with a maximum of 6 discrete values because
## more than 6 becomes difficult to discriminate; you have 32. Consider
## specifying shapes manually if you must have them.
## Warning: Removed 8197 rows containing missing values (geom_point).
```



En el caso donde cada categoría de puntos (estados) corresponde a un color la gráfica no presenta problema alguno. Sin embargo al considerar que cada estado represente una forma distinta, obtenemos **The shape palette can deal with a maximum of 6 discrete values because more than 6 becomes difficult to discriminate; you have 32. Consider specifying shapes manually if you must have them.Removed 8197 rows containing missing values (geom_point).** Lo que nos indica que en el caso de la opción **shape** solo disponemos de seis categorias.

7. ¿Qué gráficas generan los siguientes códigos?

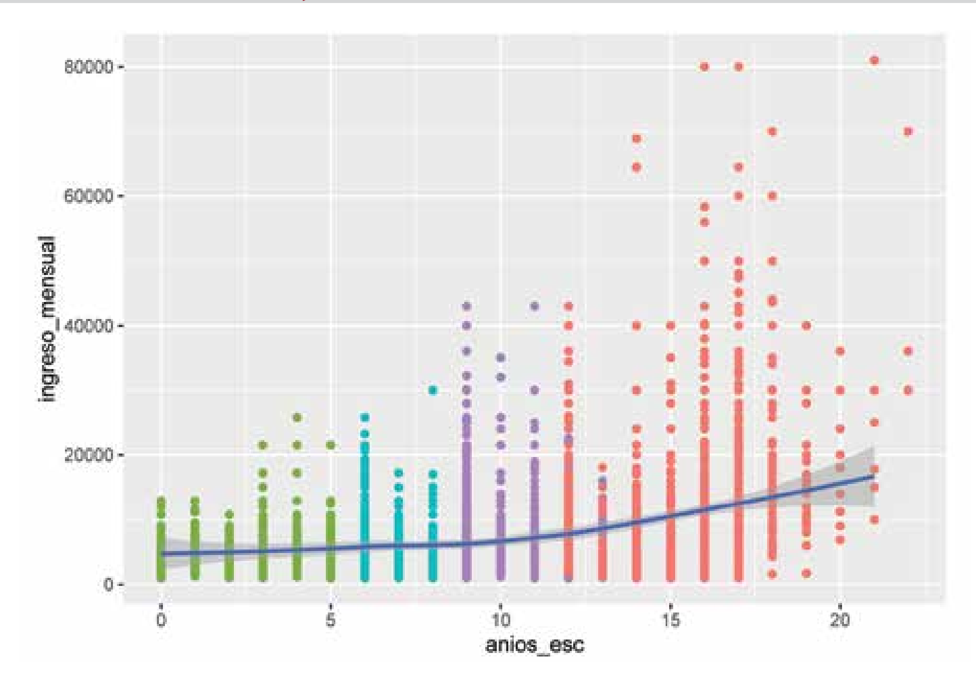
- `ggplot(data=enoe)+ geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))+ facet_grid(sex~.)`
- `ggplot(data=enoe)+ geom_point(mapping = aes(x =anios_esc, y =ingreso_mensual))+ facet_grid(. ~ tipo_empleo)`



• ¿Que función tiene "." dentro de código? Nos permite indicar se trata de única variable sobre la cual deseamos efectuar la separación de la gráfica en las diferentes sub-gráficas.

8. Replica la gráfica para separa Jalisco, pero elimina la opción se. ¿Que función tiene se?

```
ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_point(mapping = aes(color=niv_edu), show.legend = FALSE)+
     geom_smooth(
          data=filter(enoe, estado=="Jalisco"))
```



Con la indicación se podemos decidir si se incluye o no dentro de la gráfica un intervalo de confianza sobre el comportamiento esperado de la linea que ajusta a los datos.

9. ¿Cuál sería el código necesario para construir cada una de las siguientes gráficas?

En el código siguiente a cada una de las gráficas se le ha asignado un nombre de objeto dentro del entorno de **R**. Observa cuidadosamente las diferencias entre incluir una estética dentro o fuera de la función **aes**. La indicación **size=.25** al no estar dentro de la estética aes indica el tamaño de los puntos mostrados. Al final del código se muestra **grid.arrange(p1, p2, p3,p4, p5, p6)** una opción que permite ordenar en un arreglo las 6 gráficas creadas. El comando grid.arrange se encuentra dentro de la libreria **gridExtra**, por lo que tendrás que instalarla y activarla.

```
p1←ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_point(size=.25)+
     geom_smooth(se=FALSE)

p2←ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_point(size=.25)+
     geom_smooth(mapping = aes(linetype=sex),se=FALSE, show.legend = FALSE)

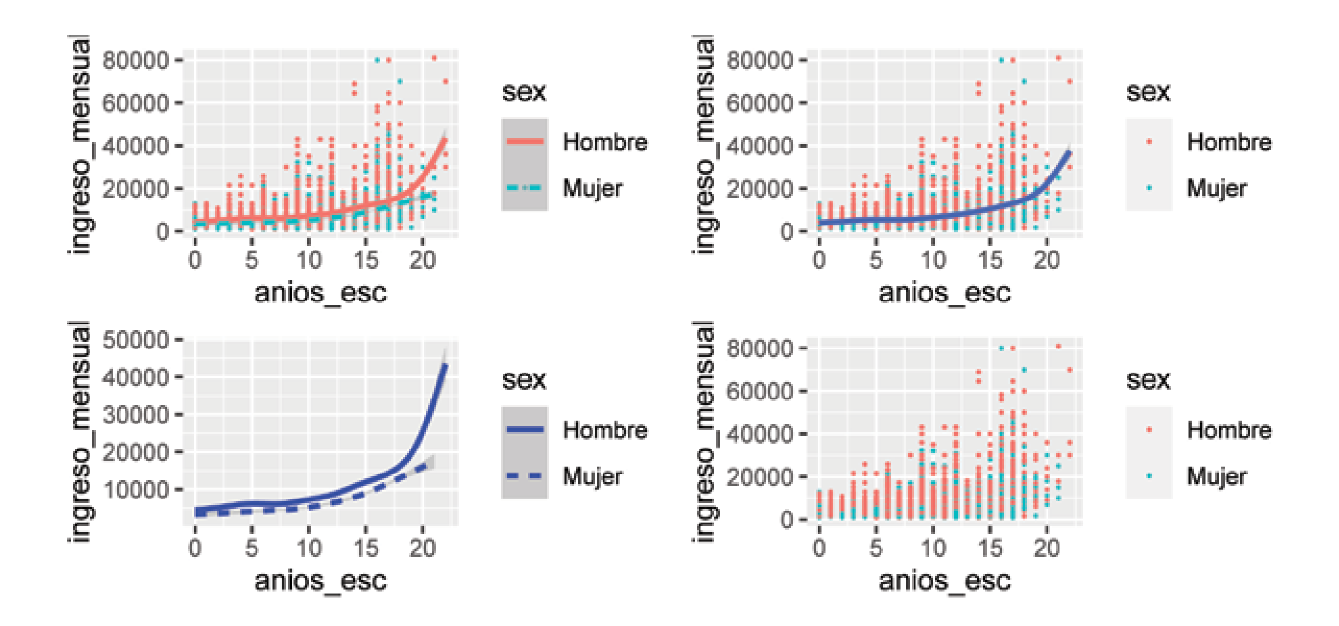
p3←ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_point(mapping = aes(color=sex),size=.25)+
     geom_smooth(mapping = aes(linetype=sex, color=sex))

p4←ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_point(mapping = aes(color=sex),size=.25)+
     geom_smooth()

p5←ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_smooth(mapping=aes(linetype=sex), color="blue")

p6←ggplot(data=enoe, mapping = aes(x =anios_esc, y =ingreso_mensual))+
     geom_point(mapping = aes(color=sex),size=.25)

grid.arrange(p1, p2, p3,p4, p5, p6)
```



10. ¿Cuál es la diferencia entre geom_col y geom_bar?

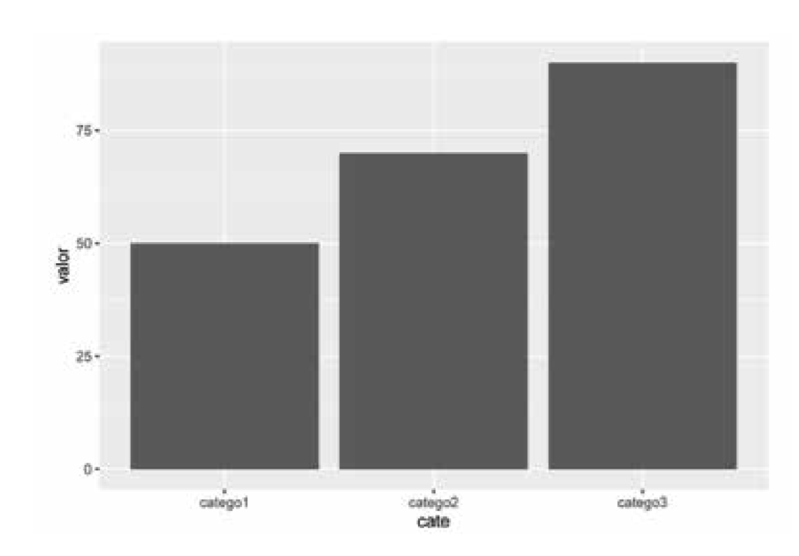
Como ya sabemos **geom_bar** usa la transformación estadistica **stat_count()** mientras que **geom_col** usa **stat_identity.** Para mostrarlo consideremos el ejemplo con los siguientes datos.

```
prueba<-tribble(
     ~cate, ~valor,
     "catego1", 50,
     "catego2", 70,
     "catego3", 90
)
prueba<-as.data.frame(prueba)
prueba
## cate valor
## 1 catego1 50
## 2 catego2 70
## 3 catego3 90
```

Por ahora, no te preocupes sobre se generaron estos datos, lo importante únicamente es su uso.

En este caso ya tenemos el total de elementos en cada categoría, por lo cual podemos usar directamente **geom_col** sin indicar ninguna transformación estadística.

```
ggplot(data=prueba)+
     geom_col(mapping = aes(x=cate, y=valor))
```

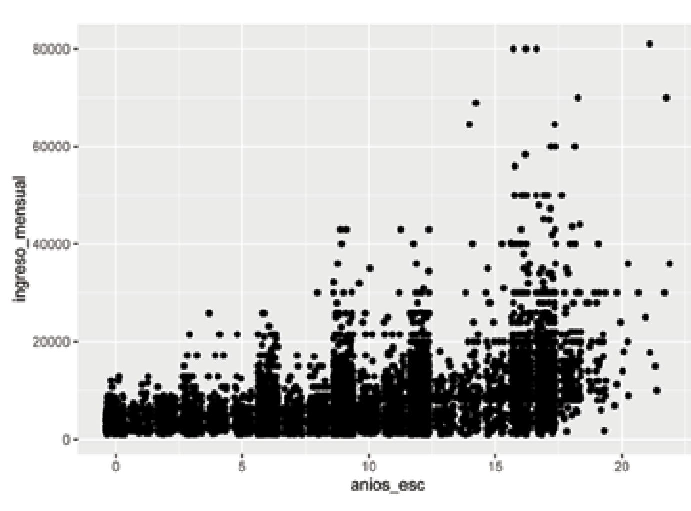
Es de notar que si indicamos

```
ggplot(data=prueba)+
     geom_bar(mapping = aes(x=cate, y=valor))
```

R nos regresará un error indicando *Error: stat_count() can only have an x or y aesthetic,* pues recordemos que la función predeterminada de **geom_bar es stat_count()**, misma que requiere una sola variable sobre la cual hacer el conteo.

11. Una de las geometrias en ggplot2 es geom_jitter, la cual crea el mismo efecto que position=" jitter" ¿Cómo sería el código para construir la misma gráfica usando geom_jitter en lugar de position="jitter"?

```
ggplot(data = enoe) +
     geom_jitter(mapping = aes(x=anios_esc, y =ingreso_mensual ))
```



Este código resulta equivalente a

- ggplot(data = enoe) + geom_point(mapping = aes(x=anios_esc, y =ingreso_mensual), position = "jitter")

Soluciones al Capítulo 4 | Transformación y exploración de datos

Contents

No olvides cargar tus datos y activar la librería 'tidyverse´

1. Utiliza el data frame enigh para crear un nuevo conjunto de datos con las siguientes características:

- Hogares de jefatura femenina menores de 30 años
- Hogares de jefatura masculina menores de 30 años
- Hogares de unipersonales menores de 30 años (los hogares unipersonales corresponden a la clave 1 de la variable "clase_hog")

```
# Hogares de jefatura femenina menores de 30 años
hogares_fem30 <- filter(enigh, sexo_jefe==2, edad_jefe<30)

# Hogares de jefatura masculina menores de 30 años
hogares_mas30 <- filter(enigh, sexo_jefe==1, edad_jefe<30)

# Hogares de unipersonales menores de 30 años
hogares_uniper30 <- filter(enigh, clase_hog==1, edad_jefe<30)
```

En cada caso hemos seleccionado un filtro que se adapte a las indicaciones, hemos usado la función **filter()**. Observa que en todos los casos el filtro establece una única condición sobre diferentes variables

2. Genera un nuevo data frame con las siguientes variables: ubica_geo, edad_jefe, tot_integ, gasto_mon, alimentos, vesti_calz, vivienda, limpieza, salud, transporte, educa_espa, personales y transf_gas. Donde las variables; alimentos, vesti_calz, vivienda, limpieza, salud, transporte, educa_espa, personales y transf_gas son rubros de gasto que componen al gasto monetario (gasto_mon). Con este nuevo conjunto de datos, calcula lo siguiente:

- El porcentaje que representa cada rubro respecto al **gasto** monetario total. Da el nombre de gastos a este data frame.
- Encuentra gráficamente a qué edad del jefe del hogar se alcanza el máximo gasto per cápita en salud

Primero seleccionamos las variables que son de interés. Esto lo hacemos con la función **select()**. Después usamos mutate() para construir las variables que necesitamos. En todos los casos se solicita expresar el gasto de cierto rubro, como porcentaje del gasto total. A estas variables les hemos asignado un nombre con la indicación **porcentaje**

```
# El porcentaje que representa cada rubro respecto al gasto monetario total
gastos <- select(enigh, ubica_geo, edad_jefe, tot_integ,
           gasto_mon, alimentos, vesti_calz, vivienda,
           limpieza, salud,transporte, educa_espa,
           personales, transf_gas) %>%
      mutate(
         alimentos_porcentaje=alimentos/gasto_mon,
         vesti_calz_porcentaje= vesti_calz/gasto_mon,
         vivienda_porcentaje= vivienda/gasto_mon,
         limpieza_porcentaje= limpieza/gasto_mon,
         salud_porcentaje= salud/gasto_mon,
         transporte_porcentaje= transporte/gasto_mon,
         educa_espa_porcentaje= educa_espa/gasto_mon,
         personales_porcentaje= personales/gasto_mon,
         transf_gas_porcentaje= transf_gas/gasto_mon,
      ) %>%
         filter (gasto_mon>0)
```

Puedes observar que ahora existe un nuevo data frame de nombre gastos, el cual contiene las siguientes variables;

```
names(gastos)
```

```
## [1] "ubica_geo"       "edad_jefe"       "tot_integ"
## [4] "gasto_mon"       "alimentos"       "vesti_calz"
## [7] "vivienda"        "limpieza"        "salud"
## [10] "transporte"     "educa_espa"      "personales"
```

```
## [13] "transf_gas"         "alimentos_porcentaje" "vesti_calz_porcentaje"
## [16] "vivienda_porcentaje" "limpieza_porcentaje" "salud_porcentaje"
## [19] "transporte_porcentaje" "educa_espa_porcentaje" "personales_porcentaje"
## [22] "transf_gas_porcentaje"
```

Para encontrar gráficamente la edad del jefe del hogar a la que se alcanza el máximo gasto per cápita en salud, es necesario primero calcular este gasto en términos per capita, es decir en relación al total de integrantes del hogar. Para ello filtramos el data frame de gastos y nos quedamos únicamente con aquellos hogares que reporten un gasto mayor a cero en salud. Después creamos una nueva variable que represente el gasto per cápita. Paso seguido lo agrupamos por la edad del jefe y promediamos. Hemos incluido también el número de hogares en cada grupo de edad.

```
salud <- filter(gastos, salud>0) %>%
          mutate(salud_capita=salud/tot_integ) %>%
          group_by(edad_jefe) %>%
          summarise(gasto_salud_capita=mean(salud_capita), hogares=n())
```

Podemos observar el data frame de salud y ver que tiene solo tres variables, la edad del jefe, el promedio del gasto per cápita en salud del hogar y el número de hogares que tienen esa característica.

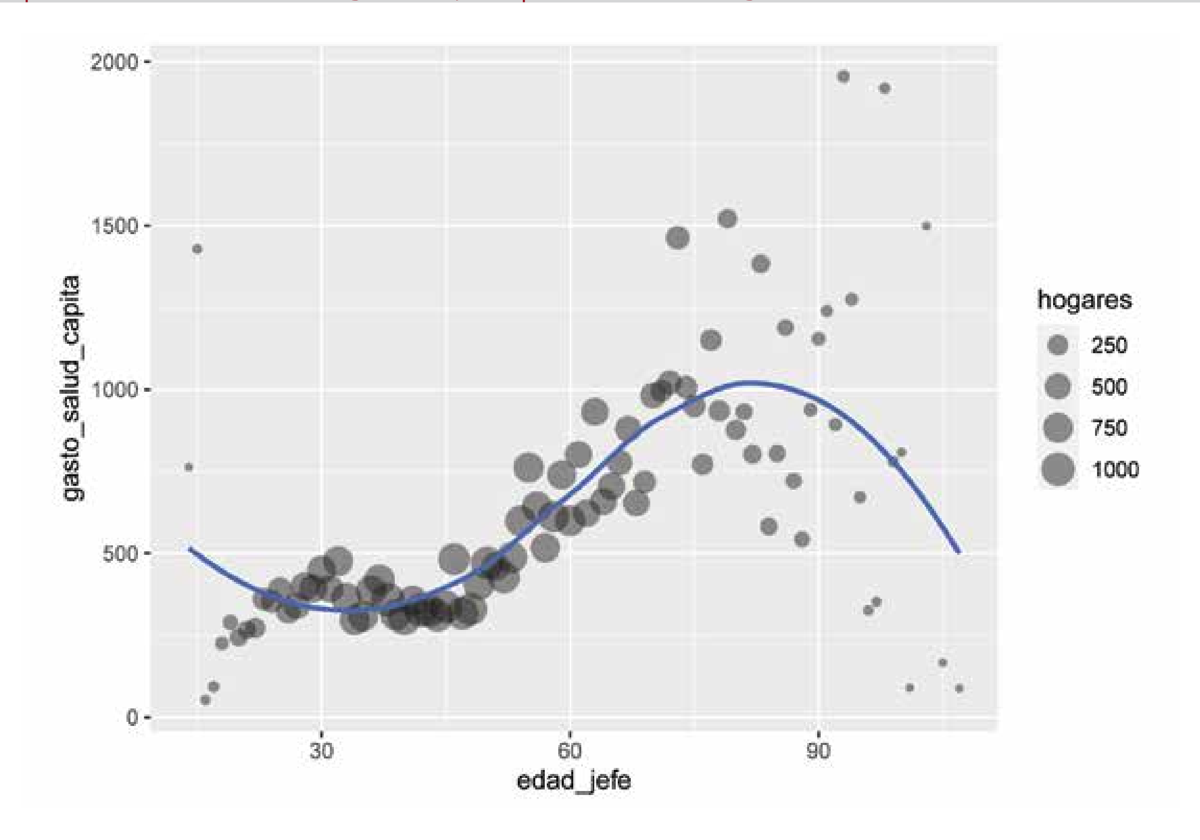
```
salud
## # A tibble: 91 x 3
##    edad_jefe  gasto_salud_capita   hogares
##    <dbl>                  <dbl>    <int>
## 1  14                     763.     1
## 2  15                     1428.    3
## 3  16                     52.3     5
## 4  17                     93.1     14
## 5  18                     225.     47
## 6  19                     289.     79
## 7  20                     244.     143
## 8  21                     266.     159
## 9  22                     272.     224
## 10 23                     360.     282
## # ... with 81 more rows
```



```
dim(salud)
## [1] 91 3
```

Finalmente, efectuamos la gráfica, la cual incluye la combinación de dos funciones geométricas, según vimos en el capítulo 3.

```
ggplot(data=salud, mapping = aes(x=edad_jefe, y=gasto_salud_capita)) +
geom_point(aes(size = hogares), alpha = 1/3) + geom_smooth(se=FALSE)
```



De manera gráfica se puede observar que la edad de mayor gasto ronda los 80 años.

3. Haciendo uso del data frame gastos generado en 2, obtén una tabla por entidad federativa donde se muestren tres cosas; A) el gasto promedio per cápita en salud, B) el promedio del gasto en salud como porcentaje del gasto monetario y C) el gasto monetario promedio. Con la información obtenida, ¿Cuál es el estado con mayor gasto per cápita en salud?

Primero es necesario identificar a donde pertenece cada una de las observaciones. Para ello usamos **substr()** para extraer la entidad federativa, la cual se encuentra presente en cada observación. Seguido calculamos la variable A) haciendo **salud_capita=salud/tot_integ**, para calcular B) solo necesitamos el promedio de la variable salud_porcentaje la cual fue previamente construida en la actividad 2. Para C) necesitamos el gasto promedio por entidad federativa **mean(gasto_mon)**.

```
salud_entidades <- mutate(gastos,
                  cve_ent=substr(ubica_geo,1,2),
                  salud_capita=salud/tot_integ) %>%
            filter(salud>0) %>%
            group_by(cve_ent) %>%
            summarise(gasto=mean(gasto_mon),
                  gasto_salud_porcentaje=mean(salud_porcentaje*100),
                  gasto_salud_capita=mean(salud_capita)
                  )
salud_entidades
```

```
## # A tibble: 32 x 4
##    cve_ent   gasto    gasto_salud_porcentaje    gasto_salud_capita
##    <chr>     <dbl>         <dbl>                <dbl>
## 1  01        40810.        4.47                 760.
## 2  02        39073.        3.15                 531.
## 3  03        44375.        3.53                 764.
## 4  04        36043.        4.49                 620.
## 5  05        37577.        3.79                 506.
## 6  06        34738.        3.62                 510.
## 7  07        21317.        4.65                 306.
## 8  08        35498.        4.28                 712.
## 9  09        47549.        3.37                 621.
## 10 10        31572.        4.90                 557.
## # ... with 22 more rows
```

Los resultados nos permiten analizar las diferencias en el gasto en salud a a través de las diferentes entidades federativas. Por ejemplo, un hogar de la entidad 1 (Aguascalientes), tiene un gasto promedio trimestral de **aproximadamente 40,810 pesos. Un 4.47% de ese gasto se dedica a cuestiones de salud**. Este significa que en promedio cada persona que vive en este estado gasta 760 pesos en este rubro.

Para obtener el estado con mayor gasto per cápita en salud, necesitamos ordenar el data frame. Para ello hacemos;

```
arrange(salud_entidades, desc(gasto_salud_capita))
```

```
## # A tibble: 32 x 4
##    cve_ent   gasto    gasto_salud_porcentaje    gasto_salud_capita
##    <chr>     <dbl>                   <dbl>      <dbl>
## 1  03        44375.                  3.53       764.
```



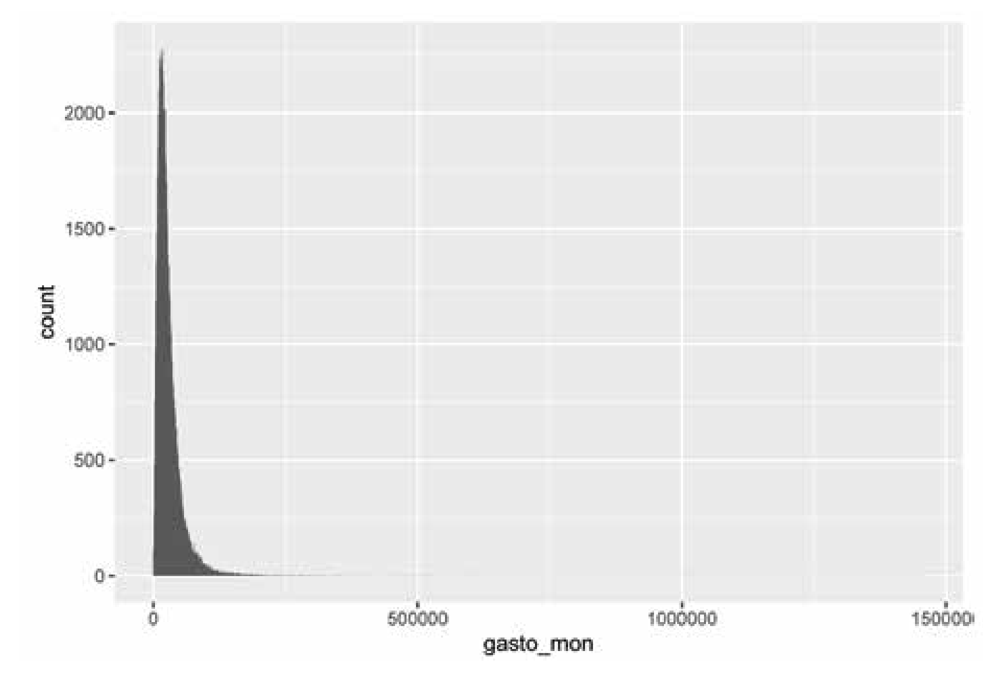
```
## 2  01         40810.             4.47      760.
## 3  14         40240.             4.90      760.
## 4  08         35498.             4.28      712.
## 5  24         31650.             5.42      652.
## 6  32         31560.             5.02      623.
## 7  09         47549.             3.37      621.
## 8  04         36043.             4.49      620.
## 9  18         35871.             4.04      611.
## 10 19         40366.             3.77      603.
## # ... with 22 more rows
```

Observamos que se trata de la entidad 3, que según las claves del INEGI es Baja California Sur.

4. Realiza un histograma del gasto monetario (gasto_mon) de los hogares + Identifica valores atípicos + Reemplaza los valores atípicos como valores perdidos

Lo primero que haremos será efectuar un histograma, para darnos cuenta de los posibles valores atípicos de la distribución. Incluimos un resumen de la variable y nos damos cuenta que existen 0, así como un valor máximo de casi 1.5 millones de pesos.

```
ggplot(data=enigh)+
     geom_histogram(mapping = aes( x= gasto_mon), binwidth = 1000)
```

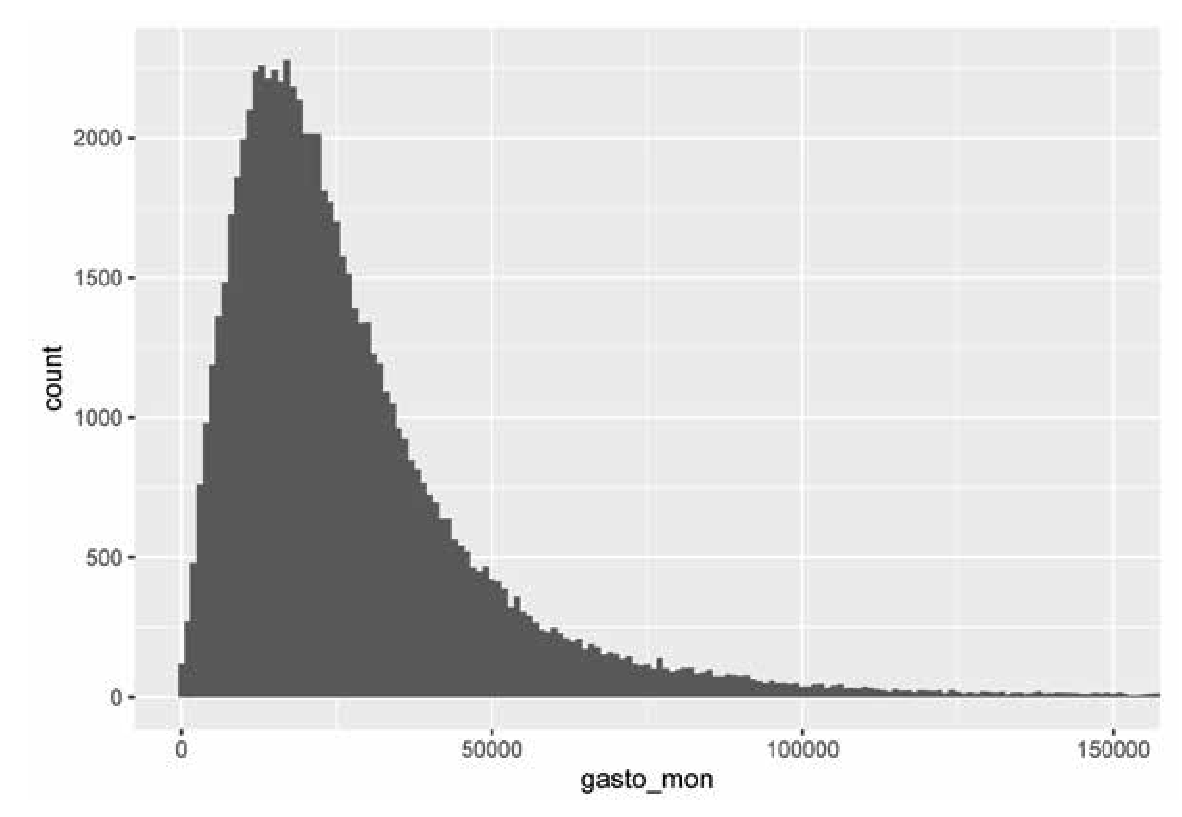



```
summary(enigh$gasto_mon)
```

```
##    Min. 1st Qu.   Median    Mean    3rd Qu.   Max.
##    0    13436     22116     28990   35428     1457729
```

La gráfica nos muestra que hay valores extremadamente grandes, los cuales podemos eliminar con el fin de tener una mejor idea del comportamiento del gasto. Para seleccionar cuales son los valores muy grandes, hacemos un corte en el histograma;

```
ggplot(data=enigh)+
     geom_histogram(mapping = aes( x= gasto_mon), binwidth = 1000)+
     coord_cartesian(xlim=c(0, 150000))
```



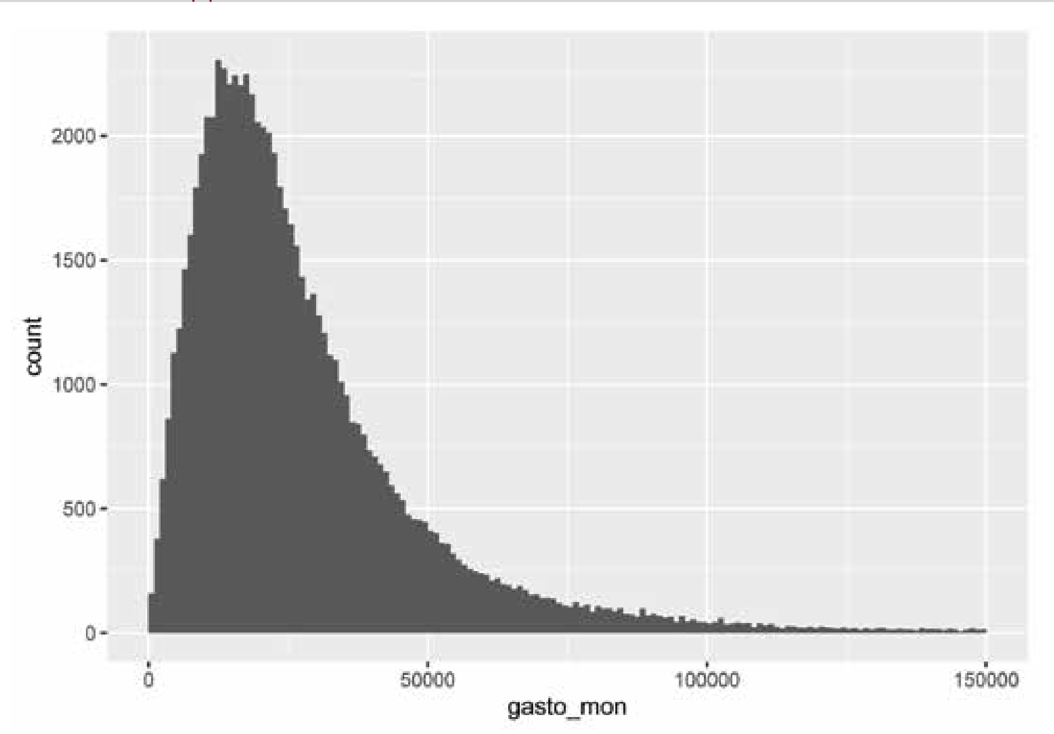
Observa que no hemos modificado los datos, únicamente estamos tratando de encontrar un corte adecuado, para presentar los datos.

Observa también qué, aunque no en gran cantidad de cero, vemos que sí hay hogares que reportan este nivel de gasto. Excluiremos estos hogares, así como los hogares que tienen gastos mayores a 150000. Podemos hacer esto dentro del mismo data frame de la **enigh**. Nota que después hemos efectuado un resumen de la variable para asegurarnos que se han efectuado los cambios.

```
enigh <- enigh %>%
mutate(gasto_mon = ifelse(gasto_mon==0 | gasto_mon>150000 , NA, gasto_mon))
summary(enigh$gasto_mon)
##    Min.  1st Qu.   Median    Mean   3rd Qu.   Max.     NA's
##    75    13398     21983     27587  34996     149957   612
```

Ahora si podemos graficar el histograma, una vez que hemos lidiado con esos valores atípicos.


```
ggplot(data=na.omit(enigh)) +
  geom_histogram(mapping = aes(x = gasto_mon), binwidth = 1000, boundary = 0)
```



Observa que los valores atípicos que identificamos, se transformaron en NA, los cuales siguen en la base, simplemente hemos decidido ignorarlos para el análisis.

5. Cálcula el gasto monetario per cápita

- Analiza la distribución del gasto per cápita de los hogares por sexo del jefe del hogar

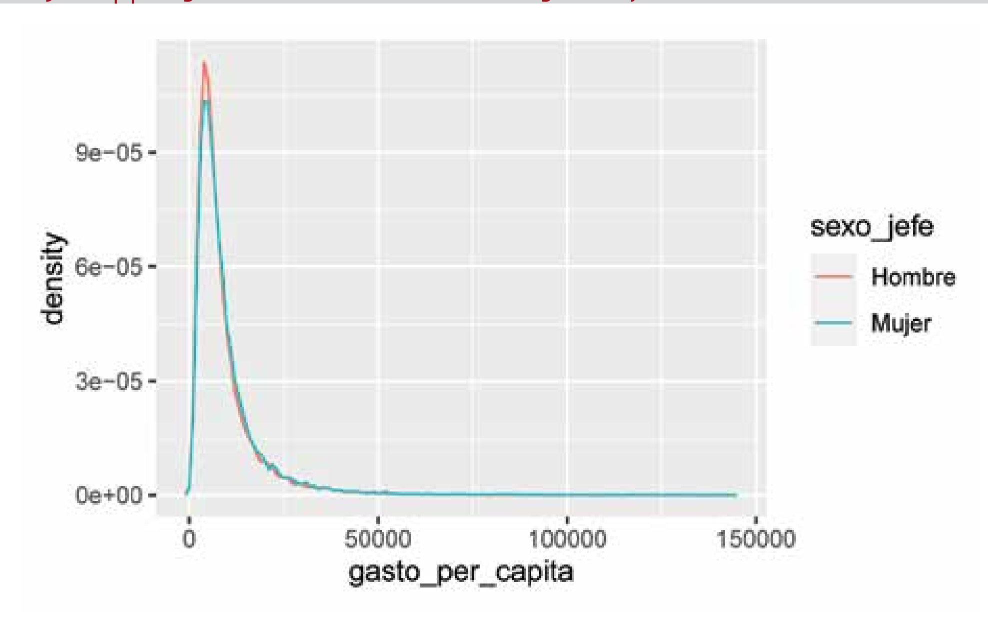
Primero construimos la variable de gasto monetario per cápita

```
enigh <-enigh %>%
mutate(gasto_per_capita = gasto_mon/tot_integ)
```

Después efectuamos el histograma correspondiente. Recuerda qué para hacer esta gráfica, la variable sexo_jefe debe ser categórica. Si no lo es hacemos;

```
enigh$sexo_jefe<-gsub(1,"Hombre", enigh$sexo_jefe)
enigh$sexo_jefe<-gsub(2,"Mujer", enigh$sexo_jefe)
```

```
ggplot(data = na.omit(enigh),
mapping = aes(x = gasto_per_capita, y = ..density..)) +
geom_freqpoly(mapping = aes(color = sexo_jefe), binwidth = 1000)
```



Soluciones al Capítulo 5 | Importación de datos

1. Identifica el error en la lectura de cada uno de los siguientes vectores

- `read_csv("a,b\n1,2,3\n4,5,6")`

En el primer caso al efectuar el código tenemos que en el caso de las filas 1 y 2, estamos solicitando tres columnas. Sin embargo, en la primera línea indicamos únicamente dos valores **a,b**. Recordando que la primera línea corresponde al nombre de las variables, la función encuentra una inconsistencia. El resultado final es un **data frame** que contiene únicamente, las observaciones que fue posible acomodar.

```
read_csv("a,b\n1,2,3\n4,5,6")
##    Warning: 2 parsing failures.
##    row col    expected actual       file
##    1     -- 2 columns 3 columns literal data
##    2     -- 2 columns 3 columns literal data
## # A tibble: 2 x 2
##    a           b
##    <dbl>       <dbl>
## 1  1           2
## 2  4           5
```

Para corregir ese error debemos hacer


```
read_csv("a,b, \n1,2,3\n4,5,6")
## Warning: Missing column names filled in: 'X3' [3]
## # A tibble: 2 x 3
##          a          b          X3
##          <dbl>      <dbl>      <dbl>
## 1        1          2          3
## 2        4          5          6
```

De esta forma existe la tercer columna y como no tiene nombre declarado, se le asigna X3

- `read_csv("a,b,c\n1,2\n1,2,3,4")`

```
read_csv("a,b,c\n1,2\n1,2,3,4")
## Warning: 2 parsing failures.
##  row col      expected    actual     file
##    1 -- 3 columns   2 columns  literal data
##    2 -- 3 columns   4 columns  literal data
## # A tibble: 2 x 3
##    a     b     c
##    <dbl>      <dbl>      <dbl>
## 1  1          2          NA
## 2  1          2          3
```

En este caso la primera línea nos indica que hay tres columnas. La primera fila contiene únicamente dos valores, mientras que la segunda incluye 4.

Observa que el **data frame** si es creado, sin embargo, no contiene todos los números. Podemos corregir este error de la siguiente forma

```
read_csv("a,b,c,\n1,2, ,\n1,2,3,4")
## Warning: Missing column names filled in: 'X4' [4]
## # A tibble: 2 x 4
##    a          b          c          X4
##    <dbl>      <dbl>      <dbl>      <dbl>
## 1  1          2          NA         NA
## 2  1          2          3          4
```

Ahora el *warning* que nos muestra indica que tenemos una columna sin nombre, la cual etiqueta de manera automática como **X4**

- `read_csv("a,b\n\"1")`

En este caso tenemos dos columnas, la primea fila sin información, mientras que en la segunda un solo dato.

También se observa que se omite un indicador cambio de fila.

```
# Problema dado
read_csv("a,b\n\"1")
```

```
## Warning: 2 parsing failures.
## row col                  expected   actual       file
##    1 a closing quote at end of file       literal data
##    1 -- 2 columns               1 columns literal data
## # A tibble: 1 x 2
##    a          b
##    <dbl>      <chr>
## 1  1          <NA>
```

```
#Solución
read_csv("a,b\n,,\n1,")
```

```
## Warning: 1 parsing failure.
## row col expected   actual     file
##    1 -- 2 columns   3 columns  literal data
## # A tibble: 2 x 2
##    a          b
##    <dbl>      <lgl>
## 1  NA         NA
## 2  1          NA
```

- `read_csv("a,b\n1,2\na,b")`

En este caso no existe complicación alguna en el código, ya que la indicación cuenta con dos columnas, y en cada fila hay dos datos.

```
read_csv("a,b\n1,2\na,b")
```



```
## # A tibble: 2 x 2
##    a          b
##    <chr>      <chr>
## 1  1          2
## 2  a          b
```

- `read_csv("a;b\n1;3")`

Observa cuidadosamente que en este caso se ha usado ; **(punto y coma)** en lugar de , **(coma)** para indicar la separación de cada columna. Cuando ha cambiado el separador de columna, debemos usar **read_delim** indicando el parámetro que hace las veces de separador.

```
# Problema dado
read_csv("a;b\n1;3")
```

```
## # A tibble: 1 x 1
## `a;b`
## <chr>
## 1 1;3
```

```
#Solución
read_delim("a;b\n1;3", delim = ";")
```

```
## # A tibble: 1 x 2
## a b
## <dbl> <dbl>
## 1 1 3
```

2. ¿Qué sucede si fijamos la configuración **decimal_mark()** y **grouping_mark()** en el mismo vector?

Para entenderlo creemos un vector como el siguiente

```
x<-"$442,185.895.145"
```

Recuerda que **grouping_mark = "."** nos permite identificar la marca de agrupación, mientras que **decimal_mark = ","** lo utilizamos para indicar el parámetro que corresponde con el punto decimal. Si usamos ambas notaciones, tendremos

```
parse_number(x, locale = locale(grouping_mark = "."))
```

```
## [1] 442.1859
```

```
parse_number(x, locale = locale(grouping_mark = ","))
## [1] 442185.9
```

```
parse_number(x, locale = locale(decimal_mark = ","))
## [1] 442.1859
```

```
parse_number(x, locale = locale(decimal_mark = "."))
## [1] 442185.9
```

```
parse_number(x, locale = locale(grouping_mark = ".", decimal_mark = "," ))
## [1] 442.1859
```

```
parse_number(x, locale = locale(grouping_mark = ",", decimal_mark = "." ))
## [1] 442185.9
```

Observa cuidadosamente el efecto que tiene cada indicación **locate** sobre el resultado deseado.

3. Construye el siguiente **data frame**. Usa **tibble**. Después crea un objeto que contenga únicamente la variable total.

educación	hogar	total
primaria	familiar	5000
secundaria	jefe mujer	2300
primaria	jefe mujer	4600

Usando la función **tibble**, podemos construir el data frame solicitado haciendo

```
df2← tibble(
     `educación`=c("primaria", "secundaria", "primaria"),
     hogar=c("familiar", "jefe mujer", "jefe hombre"),
     total=c(5000, 2300, 4600))
df2
## # A tibble: 3 x 3
##    educación  hogar            total
##    <chr>      <chr>            <dbl>
```



```
## 1  primaria   familiar         5000
## 2  secundaria jefe mujer       2300
## 3  primaria   jefe hombre      4600
```

Para crear un objeto que contenga solo la información de la variable total, hacemos

```
df3← df2 %>% .$total
df3
## [1] 5000 2300 4600
```

4. Dentro de las bases de datos de ejemplo que están disponibles en **readr** se encuentra la base de nombre **massey-rating.txt**.

• Usa la función **read_csv** para importar el archivo. ¿Qué errores detectan en la importación? Usa el comando **View()** para visualizar completamente el contenido de la base.

• Ademas del formato csv, en ocasiones es común encontrar bases de datos en formato txt. Para leer este tipo de archivos usamos la indicación **read_table**. Usa esta función para importar el archivo. ¿Mejoró la importación?

```
base←read_csv(readr_example("massey-rating.txt"))
##
## -- Column specification ------------------------------------------------
-----
## cols(
## `UCC PAY LAZ KPK RT COF BIH DII ENG ACU Rank Team Conf` = col_character()
## )
```

```
View(base)
## Observa que al ejecutar View(base), en la parte superior junto a la pesta-
ña de tu script
#te aparece un pestaña con el nombre de la base. En ese lugar puedes observar
completamente la información.
```

El principal problema de la importación es qué en una sola columna, se cargaron todos los datos correspondientes a la información de todas las variables para cada observación. La dimensión de la base es de 10x1 (10 filas una columna) Podemos darnos cuenta de esto haciendo

```
dim(base)
## [1] 10 1

base[5,1]
## # A tibble: 1 x 1
##   `UCC PAY    LAZ   KPK   RT    COF   BIH   DII   ENG  ACU Rank    Team Conf`
##   <chr>
## 1 6     6     6     5     5     7     6     5     6     11 5  Michigan St B10
```

dim(base) nos regresa el tamaño de la base, mientras que con **base[5,1]** estamos solicitando muestre la fila 5 de la columna 1.

Para corregir el problema con la importación, hacemos uso de **read_table**

```
base2<-read_table(readr_example("massey-rating.txt"))
##
## -- Column specification --------------------------------------------------
-----
##   cols(
##   UCC = col_double(),
##   PAY = col_double(),
##   LAZ = col_double(),
##   KPK = col_double(),
##   RT  = col_double(),
##   COF = col_double(),
##   BIH = col_double(),
##   DII = col_double(),
##   ENG = col_double(),
## ACU = col_double(),
## Rank = col_double(),
## Team = col_character(),
## Conf = col_character()
## )

dim(base2)
```



```
## [1] 10 13
base2[5,1]
## # A tibble: 1 x 1
##   UCC
##   <dbl>
## 1 6
```

Ahora tenemos 10 filas y 13 columnas y al solicitar **base2[5,1]** nos mostrará un solo número correspondiente al que se encuentra en la fila 5, columna 1.

Finalmente, para determinar los elementos de la lista propuesta que están no forman parte de la variable **teams** hacemos;

```
# Crear el objeto con la información de los equipos
equipos<- base2%>% .$Team
# Dar de alta la lista
lista<-c("TCU","Georgia Tech","Mississippi","Baylor","Michigan St","Ohio")
# Ejecutar parse_factor para corroborar
parse_factor(lista, levels = equipos)
## Warning: 1 parsing failure.
## row col expected actual
## 6  -- value in level set Ohio
## [1] TCU                Georgia Tech Mississippi Baylor Michigan St
## [6] <NA>
## attr(,"problems")
## # A tibble: 1 x 4
##   row col expected actual
##   <int> <int> <chr> <chr>
## 1  6 NA value in level set Ohio
## 10 Levels: Ohio St Oregon Alabama TCU Michigan St Georgia Florida St ...
Mississippi
```

Por lo cual la palabra Ohio no pertenece al conjunto de equipos.

Cada archivo que deseemos importar a **R** tiene características particulares y específicas del formato en que se encuentra, por tanto, debemos siempre poner atención a ello. Con **readr** tenemos una variedad de opciones de importación. En este documento, puedes consultar con mayor detalle otras opciones para otros tipos de archivos [Documentación_readr] (*https://cran.r-project.org/web/packages/readr/readr.pdf)*

Soluciones al Capítulo 6 | Datos ordenados

1 Solución a las Actividades

Antes de nada, recuerda cargar las librerías que necesitamos.

Antes que nada recuerda tener disponibles los datos con las manipulaciones previas. Este procedimiento se explico a lo largo de capítulo, por lo que solo lo replicaremos.

```
datos ← read_csv("datos_banco_mundial.csv")
```

```
##
## -- Column specification ----------------------------------------------------
----------------------------
## cols(
## .default = col_character(),
## `2007` = col_double(),
## `2008` = col_double(),
## `2009` = col_double()
## )
## i Use `spec()` for the full column specifications.
```

```
#Gathering
datos ← datos%>%
gather(`1960` : `2020`, key="periodo", value = "valores")
```

```
#Spreading
```



```
datos ← datos %>%
     select(-(`serie Code`)) %>%
spread(key = "serie Name", value="valores")
```

```
#Valores perdidos
datos ← datos %>%
     mutate(`Cajeros automaticos (por cada 100.000 adultos)` = ifelse
          (`Cajeros automaticos (por cada 100.000 adultos)`==".." ,
          NA, `Cajeros automaticos (por cada 100.000 adultos)`) ) %>%
     mutate(`PIB (US$ a precios constantes de 2010)` = ifelse
          (`PIB (US$ a precios constantes de 2010)`==".." ,
          NA, `PIB (US$ a precios constantes de 2010)`) ) %>%
     mutate(`Poblacion, total` = ifelse
          (`Poblacion, total`==".." ,
          NA, `Poblacion, total`) )
```

```
#Tipo numérico
datos ← datos %>%
     mutate(periodo=as.numeric(periodo)) %>%
     mutate(`Cajeros automaticos (por cada 100.000 adultos)`=as.numeric
          (`Cajeros automaticos (por cada 100.000 adultos)`)) %>%
     mutate(`PIB (US$ a precios constantes de 2010)`=as.numeric
          (`PIB (US$ a precios constantes de 2010)`)) %>%
     mutate(`Poblacion, total`=as.numeric
          (`Poblacion, total`))
```

** 1. Utilizando el data frame *datos* ya en su forma *tidy data*, construye las siguientes variables:**

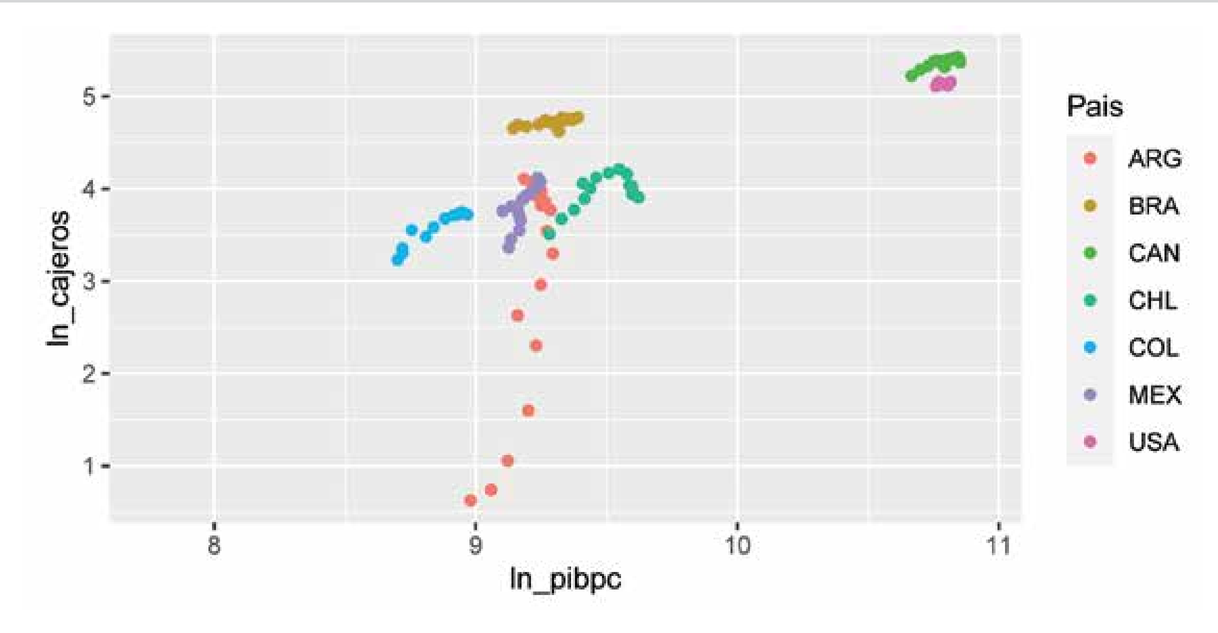
- Logaritmo natural del PIB
- PIB per cápita
- Logaritmo del PIB per cápita
- Logaritmo de los Cajeros automáticos (por cada 100.000 adultos)

Construimos las variables solicitadas haciendo uso de la función **mutate()**

```
datos ← datos %>%
     mutate(ln_pib = log(`PIB (US$ a precios constantes de 2010)`)) %>%
     mutate(pib_pc = (`PIB (US$ a precios constantes de 2010)`/`Poblacion,
     total`)) %>%
     mutate(ln_pibpc = log(pib_pc)) %>%
     mutate(ln_cajeros = log(`Cajeros automaticos (por cada 100.000 adul
     tos)`))
```

- Gráfica un diagrama de dispersión entre el logaritmo del PIB per cápita y el logaritmo del los cajeros automáticos.

```
ggplot(data = datos) +
geom_point(mapping = aes(x = ln_pibpc, y = ln_cajeros , col=Pais), na.rm
=TRUE )
```



2. Importa la tabla "importaciones_exportaciones"

- Realiza la manipulación necesaria para transformar estos datos a tipo *tidy data*

Cargamos los datos y nos damos cuenta que no son tipo *tidy*

```
comercio <- read_csv("importaciones_exportaciones.csv")
##
## -- Column specification ----------------------------------------------------
----------------------------
## cols(
## .default = col_double(),
## serie = col_character(),
## `pais nombre` = col_character(),
## pais = col_character(),
## `YR 2020` = col_character()
## )
## i Use `spec()` for the full column specifications.
```



```
comercio
## # A tibble: 8 x 64
##    serie `pais nombre` pais `YR 1960` `YR 1961` `YR 1962` `YR 1963` `YR 1964`
##    <chr> <chr>         <chr> <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1  Expo~ Colombia      COL  3.88e 9   3.62e 9   3.91e 9 3.82e 9 4.04e 9
## 2  Expo~ Brasil        BRA  7.09e 9   7.46e 9   6.91e 9 8.84e 9 7.64e 9
## 3  Expo~ México        MEX  6.50e 9   7.08e 9   7.52e 9 7.72e 9 8.26e 9
## 4  Expo~ Chile         CHL  3.32e 9   3.48e 9   3.58e 9 3.76e 9 4.11e 9
## 5  Impo~ Colombia      COL  2.36e 9   2.45e 9   2.54e 9 2.54e 9 3.16e 9
## 6  Impo~ Brasil        BRA  1.15e10   1.11e10   1.10e10 1.07e10 9.35e 9
## 7  Impo~ México        MEX  1.26e10   1.25e10   1.27e10 1.40e10 1.61e10
## 8  Impo~ Chile         CHL  3.00e 9   3.35e 9   3.10e 9 3.18e 9 3.52e 9
## # ... with 56 more variables: `YR 1965` <dbl>, `YR 1966` <dbl>, `YR
## # 1967` <dbl>, `YR 1968` <dbl>, `YR 1969` <dbl>, `YR 1970` <dbl>, `YR
## # 1971` <dbl>, `YR 1972` <dbl>, `YR 1973` <dbl>, `YR 1974` <dbl>, `YR
## # 1975` <dbl>, `YR 1976` <dbl>, `YR 1977` <dbl>, `YR 1978` <dbl>, `YR
## # 1979` <dbl>, `YR 1980` <dbl>, `YR 1981` <dbl>, `YR 1982` <dbl>, `YR
## # 1983` <dbl>, `YR 1984` <dbl>, `YR 1985` <dbl>, `YR 1986` <dbl>, `YR
## # 1987` <dbl>, `YR 1988` <dbl>, `YR 1989` <dbl>, `YR 1990` <dbl>, `YR
## # 1991` <dbl>, `YR 1992` <dbl>, `YR 1993` <dbl>, `YR 1994` <dbl>, `YR
## # 1995` <dbl>, `YR 1996` <dbl>, `YR 1997` <dbl>, `YR 1998` <dbl>, `YR
## # 1999` <dbl>, `YR 2000` <dbl>, `YR 2001` <dbl>, `YR 2002` <dbl>, `YR
## # 2003` <dbl>, `YR 2004` <dbl>, `YR 2005` <dbl>, `YR 2006` <dbl>, `YR
## # 2007` <dbl>, `YR 2008` <dbl>, `YR 2009` <dbl>, `YR 2010` <dbl>, `YR
## # 2011` <dbl>, `YR 2012` <dbl>, `YR 2013` <dbl>, `YR 2014` <dbl>, `YR
## # 2015` <dbl>, `YR 2016` <dbl>, `YR 2017` <dbl>, `YR 2018` <dbl>, `YR
## # 2019` <dbl>, `YR 2020` <chr>
```

Efectuamos la manipulaciones adecuadas, reemplazamos valores perdidos y nos aseguramos que las variables sean de tipo númerico

```
comercio <- comercio %>%
     gather(`YR 1960` : `YR 2020`, key="periodo2", value = "valores") %>%
spread(key = "serie", value="valores") %>%
     separate(periodo2, into = c("prefijo", "periodo"), sep = " ") %>%
     select(-(prefijo)) %>%
```

```
    mutate(periodo=as.numeric(periodo)) %>%
    mutate(Exportaciones = ifelse(Exportaciones==".." , NA, as.numeric(Ex
    portaciones)) ) %>%
    mutate(Importaciones = ifelse(Importaciones==".." , NA, as.numeric(Im
    portaciones)) )
comercio
```

```
## # A tibble: 244 x 5
## `pais nombre` pais  periodo    Exportaciones   Importaciones
##    <chr>        <chr> <dbl>    <dbl>           <dbl>
## 1  Brasil       BRA   1960     7090856270      11489666919
## 2  Brasil       BRA   1961     7459161437      11088857576
## 3  Brasil       BRA   1962     6906663517      10955313149
## 4  Brasil       BRA   1963     8840526105      10688135445
## 5  Brasil       BRA   1964     7643354297      9352074878
## 6  Brasil       BRA   1965     7919802822      8016014229
## 7  Brasil       BRA   1966     8840704863      10554388261
## 8  Brasil       BRA   1967     8564410064      11355997611
## 9  Brasil       BRA   1968     9945802652      14027915743
## 10 Brasil       BRA   1969     11879704501     15631135671
## # ... with 234 more rows
```

```
summary(comercio)
```

```
##    pais nombre       pais              periodo        Exportaciones
##    Length:244        Length:244        Min.  :1960     Min.  :3.321e+09
##    Class :character  Class :character  1st Qu.:1975   1st Qu.:1.130e+10
##    Mode  :character  Mode  :character  Median :1990   Median :3.655e+10
##                                        Mean  :1990     Mean  :7.914e+10
##                                        3rd Qu.:2005   3rd Qu.:9.064e+10
##                                        Max.  :2020     Max.  :4.920e+11
##                                                        NA's  :4
##    Importaciones
##    Min.         :2.357e+09
##    1st Qu.      :1.004e+10
##    Median       :3.311e+10
##    Mean         :7.598e+10
```



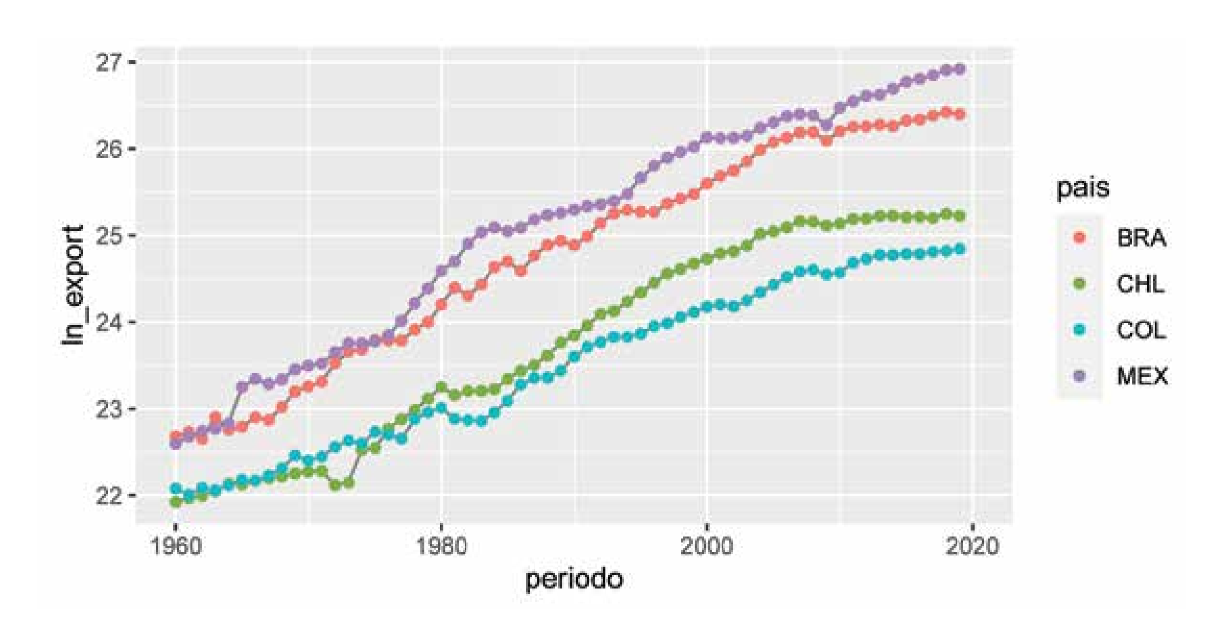
```
##    3rd Qu.      :8.229e+10
##    Max.         :4.880e+11
##    NA's         :4
```

• Calcula la diferencia (déficit comercial) entre las exportaciones e importaciones

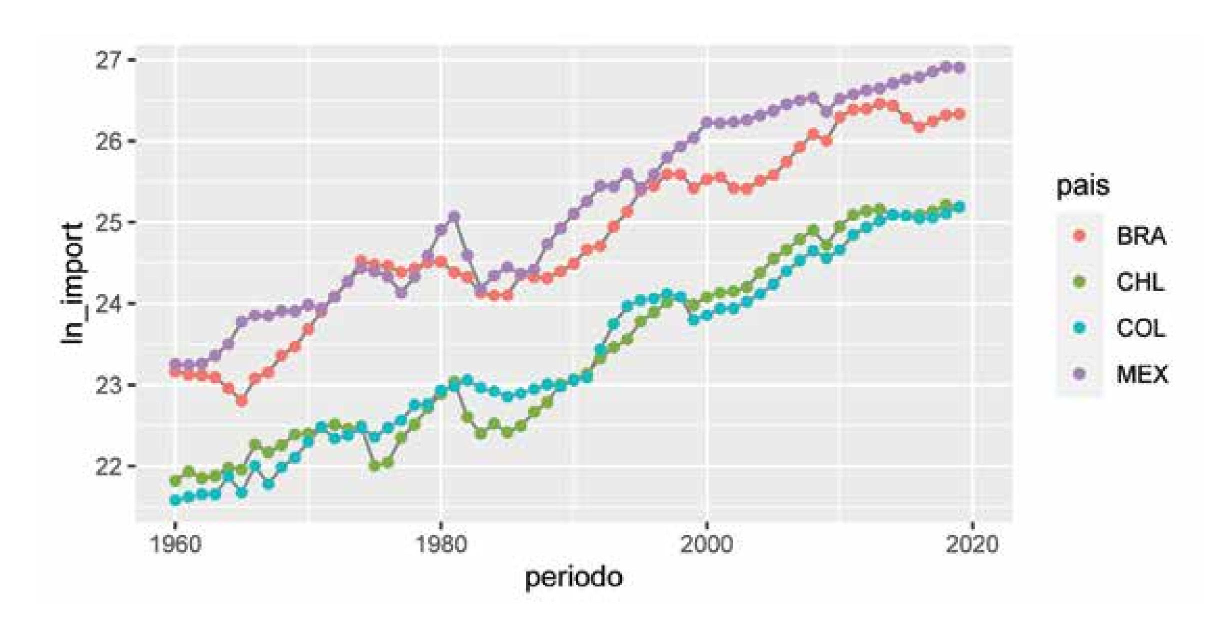
```
comercio <- comercio %>%
    mutate(ln_export = log(Exportaciones)) %>%
    mutate(ln_import = log(Importaciones)) %>%
    mutate(deficit = (Exportaciones-Importaciones))
```

• Gráfica el logaritmo del las exportaciones y las importaciones

```
ggplot(comercio, aes(periodo, ln_export))+
     geom_line(aes(group = pais), color = "grey50", na.rm = TRUE) +
geom_point(aes(color = pais), na.rm = TRUE)
```

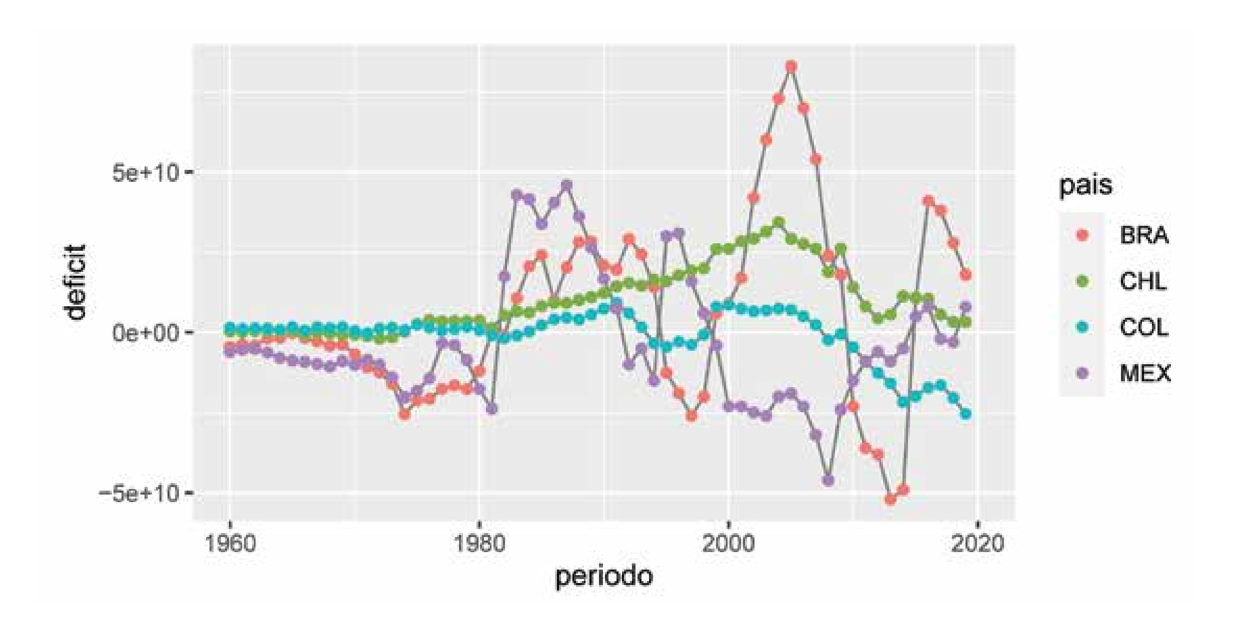



```
ggplot(comercio, aes(periodo, ln_import))+
     geom_line(aes(group = pais), color = "grey50",na.rm = TRUE) +
geom_point(aes(color = pais),na.rm = TRUE)
```



• Gráfica el déficit comercial

```
ggplot(comercio, aes(periodo, deficit))+
     geom_line(aes(group = pais), color = "grey50",na.rm = TRUE) +
geom_point(aes(color = pais),na.rm = TRUE)
```



Soluciones al Capítulo 7 | Datos Relacionales

Contents

Antes de nada, no olvides cargar tus datos y activar las librerías necesarias

```
setwd("~/Dropbox/Curso de R/Cap7_Datos_Relacionales")
library(haven)
library(tidyverse)
hogares <- read_dta("hog_jal.dta")
viviendas <- read_dta("viv_jal.dta")
personas <- read_dta("per_jal.dta")
trabajos <- read_dta("trab_jal.dta")
ingresos <- read_dta("ing_jal.dta")
```

1. Determina el promedio del ingreso corriente (ing_cor) de los hogares que se ubican en viviendas que no disponen de electricidad. (aquellas donde disp_elect=5)

Usando el diagrama de relación de la ENIGH observamos que la base de datos **viviendas** contiene la información sobre la variable **disp_elect**, mientras que la información sobre **ing_corse** encuentra en la base hogares. En consecuencia, debemos combinar ambas bases de datos, para lograr que en la base de datos de **hogares**, exista al información sobre la variable **disp_elect** . Para ello hacemos;

```
hogares %>%
     select(folioviv, foliohog, ing_cor) %>%
     left_join(viviendas, by="folioviv") %>%
```



```
     group_by(disp_elect) %>%
     summarise(mean(ing_cor))
```

```
## # A tibble: 5 x 2
##    disp_elect `mean(ing_cor)`
##    <chr>      <dbl>
## 1  1          56055.
## 2  2          17051.
## 3  3          66396.
## 4  4          35502.
## 5  5          8826.
```

Observa que primero hemos efectuado un **select** con el fin de usar solo las variables que necesitamos de la base de hogares. Después efectuar la combinación de las bases, para finalmente agrupar y obtener el promedio de **ing_cor** en cada categoría de variable **disp_elect**. Observa también que hemos usado el operador **pipe**.

2. Determina cuantos hombres y cuántas mujeres viven en viviendas que tiene un estrato socioeconómico bajo (est_socio=1)

En este caso el procedimiento es un poco mas complejo. Primero necesitamos determinar cuantos hombre y cuantas mujeres hay en cada vivienda. Para ello creamos un **data frame** que incluya solo la cantidad de hombres por vivienda y uno la cantidad de mujeres. Para esto hacemos uso de los conocimientos previos del capítulo 4.

Para los hombres hacemos;

```
hombres<-group_by(personas, folioviv, sexo) %>%
summarise(hombres=n(), .groups = "drop") %>%
     filter(sexo==1) %>%
     select(-sexo)
hombres
```

```
## # A tibble: 1,885 x 2
##    folioviv      hombres
##    <chr>         <int>
## 1  1400180001    1
## 2  1400180002    2
## 3  1400180005    1
## 4  1400189601    3
## 5  1400189604    3
```

```
## 6  1400189605       3
## 7  1400189606       4
## 8  1400264701       2
## 9  1400264702       1
## 10 1400264704       3
## # ... with 1,875 more rows
```

Para las mujeres

```
mujeres<-group_by(personas, folioviv, sexo) %>%
summarise(mujeres=n(),.groups = "drop") %>%
     filter(sexo==2) %>%
     select(-sexo)
mujeres
```

```
## # A tibble: 1,942 x 2
## folioviv mujeres
## <chr> <int>
## 1 1400180001 1
## 2 1400180002 3
## 3 1400180006 1
## 4 1400189601 3
## 5 1400189603 2
## 6 1400189604 1
## 7 1400189605 1
## 8 1400189606 4
## 9 1400264701 5
## 10 1400264702 1
## # ... with 1,932 more rows
```

Una vez que tenemos identificada la cantidad de hombres y mujeres en cada vivienda, vamos a juntar esa información en un solo **data frame**.

```
sexo_viv<-full_join(hombres, mujeres, by="folioviv")
sexo_viv
```

```
## # A tibble: 2,095 x 3
##    folioviv         hombres    mujeres
##    <chr>            <int>      <int>
```



```
## 1  1400180001       1          1
## 2  1400180002       2          3
## 3  1400180005       1          NA
## 4  1400189601       3          3
## 5  1400189604       3          1
## 6  1400189605       3          1
## 7  1400189606       4          4
## 8  1400264701       2          5
## 9  1400264702       1          1
## 10 1400264704       3          5
## # ... with 2,085 more rows
```

Vemos que en cada vivienda ya tenemos identificados cuantos hombre y cuantas mujeres hay. Para ello observa que hemos usado **full_join**. En consecuencia, en los hogares donde no existan hombre o mujeres, tendremos valores marcados con **NA**

Ahora solo debemos agregar a cada vivienda el estrato socioeconómico y agrupar la suma de hombres y mujeres en cada vivienda de cada estrato.

```
viviendas %>%
     select(folioviv, est_socio) %>%
     left_join(sexo_viv, by="folioviv") %>%
     group_by(est_socio) %>%
     summarise(sum(na.omit(hombres)), sum(na.omit(mujeres)),.groups = "drop")
```

```
## # A tibble: 4 x 3
##    est_socio  `sum(na.omit(hombres))`     `sum(na.omit(mujeres))`
##    <chr>      <int>                       <int>
## 1  1          365                         359
## 2  2          2089                        2172
## 3  3          1085                        1160
## 4  4          253                         286
```

Observa que hemos usado na.omit() para evitar que se consideren esos valores.

3. Determina en promedio cuanto gastan en salud los hogares cuya vivienda se ubica en vecindad. Esta característica se identifica cuando en la base viviendas la variable tipo_viv toma el valor de 3

La información sobre el gasto en salud se ubica en la base **hogares,** mientras que la característica sobre el tipo de vivienda, esta en la base de **viviendas**. Entonces hacemos la combinación de ambas y después solicitamos la agrupación por tipo de vivienda. Una vez agrupadas, calculamos el gasto promedio en salud:

```
hogares %>%
     select(folioviv, foliohog, salud) %>%
     left_join(viviendas, by=c("folioviv")) %>%
     group_by(tipo_viv) %>%
     summarise(total_salud=mean(salud), .groups = "drop")
```

```
## # A tibble: 6 x 2
##    tipo_viv   total_salud
##    <chr>      <dbl>
## 1  &          202.
## 2  1          1419.
## 3  2          1075.
## 4  3          264.
## 5  4          687.
## 6  5          1924.
```

De esta manera, nos damos cuenta de que hogares que viven en vecindad, gastan en promedio $264 pesos en salud.

4. Determina el promedio de gasto en educación y esparcimiento (edu_espa) de los hogares, donde todos sus miembros saben leer y escribir (alfabetism=1)

Debemos primero identificar a los hogares donde todos su miembros saben leer y escribir, para ello trabajamos con la base **personas**. Primero contaremos, cuando miembros en cada hogar, saben leer y cuantos no.

Los que saben leer y escribir:

```
si_leer<-group_by(personas, folioviv, foliohog, alfabetism) %>%
     summarise(si_leer=n(), .groups = "drop") %>%
     filter(alfabetism==1) %>%
     select(-alfabetism)
si_leer
```

```
## # A tibble: 2,108 x 3
##    folioviv   foliohog si_leer
##    <chr>      <chr>      <int>
## 1  1400180001 1          2
## 2  1400180002 1          5
## 3  1400180005 1          1
## 4  1400180006 1          1
## 5  1400189601 1          6
## 6  1400189603 1          2
## 7  1400189604 1          4
## 8  1400189605 1          4
## 9  1400189606 1          2
## 10 1400189606 2          4
## # ... with 2,098 more rows
```

Los que no saben leer y escribir:

```
no_leer<-group_by(personas, folioviv, foliohog, alfabetism) %>%
     summarise(no_leer=n(),.groups = "drop") %>%
     filter(alfabetism==2) %>%
     select(-alfabetism)
no_leer
```

```
## # A tibble: 654 x 3
##    folioviv   foliohog no_leer
##    <chr>      <chr>      <int>
## 1  1400189606 2          1
## 2  1400264706 1          1
## 3  1400289201 1          1
## 4  1400289202 1          1
## 5  1400289203 1          1
## 6  1400289204 1          1
## 7  1400507901 1          1
## 8  1400555306 1          1
## 9  1400563902 1          1
## 10 1400586804 1          1
## # ... with 644 more rows
```

Los juntamos, para tener toda la información de los hogares en un mismo data frame

```
alfa_hog<-full_join(si_leer, no_leer, by=c("folioviv", "foliohog"))
alfa_hog
```

```
## # A tibble: 2,152 x 4
##    folioviv   foliohog si_leer no_leer
##    <chr>      <chr>      <int> <int>
## 1  1400180001 1          2     NA
## 2  1400180002 1          5     NA
## 3  1400180005 1          1     NA
## 4  1400180006 1          1     NA
## 5  1400189601 1          6     NA
## 6  1400189603 1          2     NA
## 7  1400189604 1          4     NA
## 8  1400189605 1          4     NA
## 9  1400189606 1          2     NA
## 10 1400189606 2          4     1
## # ... with 2,142 more rows
```

Finalmente agregamos el gasto en educación y esparcimiento.

```
hogares %>%
     select(folioviv, foliohog, educa_espa) %>%
     left_join(alfa_hog, by = c("folioviv", "foliohog"))
```

```
## # A tibble: 2,152 x 5
##    folioviv   foliohog educa_espa si_leer no_leer
##    <chr>      <chr>    <dbl>      <int> <int>
## 1  1400180001 1        0          2     NA
## 2  1400180002 1        1452.      5     NA
## 3  1400180005 1        2903.      1     NA
## 4  1400180006 1        2961.      1     NA
## 5  1400189601 1        5205       6     NA
## 6  1400189603 1        5070       2     NA
## 7  1400189604 1        9630       4     NA
## 8 1400189605  1        11640      4     NA
```



```
## 9  1400189606 1        566.       2     NA
## 10 1400189606 2        0          4     1
## # ... with 2,142 more rows
```

Observa que ahora tenemos toda la información por hogar. En algunos casos tenemos NA, ya que en ese hogar no se identifican personas con la caraterística solicitada.

Si en el hogar la variable **no_leer=NA**, significa que en ese hogar no hay miembros que no saben leer (todos saben). Por lo cual, nos quedaremos únicamente con estos hogares. Eso lo hacemos gracias a la indicación **is.na(no_leer)=="TRUE"** que indica que conservemos únicamente las observaciones donde se cumpla la condición.

```
hogares %>%
     select(folioviv, foliohog, educa_espa) %>%
     left_join(alfa_hog, by = c("folioviv", "foliohog")) %>%
     filter(is.na(no_leer)=="TRUE")
```

```
## # A tibble: 1,498 x 5
##    folioviv   foliohog educa_espa si_leer no_leer
##    <chr>      <chr>    <dbl>      <int> <int>
## 1  1400180001 1        0          2     NA
## 2  1400180002 1        1452.      5     NA
## 3  1400180005 1        2903.      1     NA
## 4  1400180006 1        2961.      1     NA
## 5  1400189601 1        5205       6     NA
## 6  1400189603 1        5070       2     NA
## 7  1400189604 1        9630       4     NA
## 8  1400189605 1        11640      4     NA
## 9  1400189606 1        566.       2     NA
## 10 1400264701 1        794.       6     NA
## # ... with 1,488 more rows
```

Después aplicamos el promedio a estas observaciones y listo, ya tenemos el gasto promedio en educación y esparcimiento de los hogares donde todos su miembros saben leer y escribir.

```
hogares %>%
     select(folioviv, foliohog, educa_espa) %>%
     left_join(alfa_hog, by = c("folioviv", "foliohog")) %>%
     filter(is.na(no_leer)=="TRUE") %>%
```

```
    summarise(mean(educa_espa))
```

```
## # A tibble: 1 x 1
##   `mean(educa_espa)`
##   <dbl>
## 1 4901.
```

5. Determina cuantas personas habitan en viviendas propias (aquellas donde la variable tenencia=4)

Primero calculamos el total de personas por hogar

```
tot_hog<-group_by(personas, folioviv, foliohog) %>%
    summarise(tot_per_hog=n(), .groups = "drop")
tot_hog
```

```
## # A tibble: 2,152 x 3
##   folioviv foliohog tot_per_hog
##   <chr>    <chr>    <int>
## 1  1400180001 1        2
## 2  1400180002 1        5
## 3  1400180005 1        1
## 4  1400180006 1        1
## 5  1400189601 1        6
## 6  1400189603 1        2
## 7  1400189604 1        4
## 8  1400189605 1        4
## 9  1400189606 1        2
## 10 1400189606 2        6
## # ... with 2,142 more rows
```

Ahora agregamos la variable sobre la tenencia de la vivienda y conservamos las variables de interés.

```
left_join(tot_hog, viviendas, by="folioviv") %>%
    select(folioviv, foliohog, tot_per_hog, tenencia)
```

```
## # A tibble: 2,152 x 4
```



```
##   folioviv foliohog tot_per_hog tenencia
##   <chr>    <chr>    <int>       <chr>
## 1  1400180001 1        2           4
## 2  1400180002 1        5           4
## 3  1400180005 1        1           1
## 4  1400180006 1        1           4
## 5  1400189601 1        6           1
## 6  1400189603 1        2           1
## 7  1400189604 1        4           4
## 8  1400189605 1        4           1
## 9  1400189606 1        2           4
## 10 1400189606 2        6           4
## # ... with 2,142 more rows
```

Finalmente agrupamos por tipo de tenencia y contamos

```
left_join(tot_hog, viviendas, by="folioviv") %>%
    select(folioviv, foliohog, tot_per_hog, tenencia) %>%
    group_by(tenencia) %>%
    summarise(Total=sum(tot_per_hog),.groups = "drop")
```

```
## # A tibble: 6 x 2
##   tenencia Total
##   <chr>    <int>
## 1 1        1714
## 2 2        1158
## 3 3        573
## 4 4        4088
## 5 5        189
## 6 6        47
```

Los mecanismos que hemos usado para dar solución a las actividades plenteadas no son únicos. Tu podrías encontrar una forma distinta de resolverlo y en algunos casos omitir pasos en el desarrollo del código.

Soluciones al Capítulo 8 | Cadenas de Texto

Contents

Recuerda que primero es necesario cargar las librerías necesarias. En este caso debemos incluir la librería *datos*, para trabajar con el conjunto de palabras y oraciones que tiene *R* en idioma español. Recuerda que antes de activar cualquier librería es necesario instalarla.

```
library(tidyverse)
library(stringr)
library(datos)
```

1. Usa el vector de cadenas de texto palabras para determinar lo siguiente;

- Un subconjunto donde se encuentren las palabras que comiencen con h

```
str_subset(palabras, "^h")
```

```
## [1]     "ha"      "haber"    "había"    "habían"  "habla"   "hablar"
## [7]     "habrá"   "habría"   "hace"     "hacen"   "hacer"   "hacerlo"
## [13]    "hacia"   "hacía"    "haciendo" "han"     "has"     "hasta"
## [19]    "hay"     "haya"     "he"       "hecho"   "hechos"  "hemos"
## [25]    "hermano" "hicieron" "hija"     "hijo"    "hijos"   "historia"
## [31]    "hizo"    "hombre"   "hombres"  "hora"    "horas"   "hospital"
## [37]    "hoy"     "hubiera"  "hubo"     "humana"  "humano"  "humanos"
```


- Encuentra las palabras que tienen 5 o más vocales

Primero asignamos el conjunto de palabras a un *data frame*, despues contamos cuantas vocales tiene cadena (palabra). Finalmente filtramos aquellas donde el numero de vocales sea mayor o igual a 5.

```
vocales<-tibble(palabras)
vocales %>%
     mutate(
          num_voc=str_count(palabras, "[aeiou]")) %>%
     filter(num_voc>=5)
```

```
## # A tibble: 47 x 2
##    palabras         num_voc
##    <chr>            <int>
## 1  actividades      5
## 2  administración   5
## 3  asociación       5
## 4  autoridades      6
## 5  características  5
## 6  comunicación     5
## 7  conciencia       5
## 8  condiciones      5
## 9  conocimiento     6
## 10 consecuencia     6
## # ... with 37 more rows
```

- Encuentra las palabras que se ubican en la posición 560 y 625

Lo único que hay que hacer es asignar un número a cada cadena en el vector y después extraer la cadena contenida.

```
df <- tibble(
     palabra = palabras,
     i = seq_along(palabra)
     )
df[560,]
```

```
## # A tibble: 1 x 2
##    palabra    i
##    <chr>      <int>
## 1  militares  560
```

```
df[625,]
```

```
## # A tibble: 1 x 2
##    palabra    i
##    <chr>      <int>
## 1  objeto     625
```

• Toma las primeras 5 palabras (cadenas) y agrúpalas todas en una sola cadena de texto usando (/)

```
cinco<-palabras[1:5]
cinco
```

```
## [1] "a" "abril" "acción" "acciones" "acerca"
```

```
cinco %>%
     str_c(collapse = "/")
```

```
## [1] "a/abril/acción/acciones/acerca"
```

2. Usa el vector de cadenas de texto oraciones para determinar lo siguiente;

• Todas las cadenas que contienen al menos una vez una de las siguientes coincidencias; rojo, azul, amarillo, verde.

Primero debemos construir una cadena con las coincidencias que buscamos. Después buscamos el vector de coincidencias en el vector de oraciones.

```
coincidencia<-("rojo|azul|amarillo|verde")
coincide<-str_subset(oraciones, coincidencia)
coincide
```

```
## [1] "Pega la hoja en el fondo azul oscuro."
## [2] "Instalaron azulejos verdes en la cocina."
## [3] "Si arrojo la taza azul al suelo se romperá."
## [4] "Dos peces azules nadaban en el tanque."
## [5] "Una voluta de nube flotaba en el aire azul."
## [6] "Las hojas se vuelven de color marrón y amarillo en el otoño."
## [7] "La mancha en el papel secante fue hecha por la tinta verde."
## [8] "El cojín del sofá es de color rojo y de peso ligero."
```



```
## [9] "El cielo de la mañana era claro y azul brillante."
## [10] "Una grulla azul es una ave zancuda y alta."
## [11] "La lámpara brillaba con una llama verde y continua."
## [12] "La planta creció grande y verde en la ventana."
## [13] "Recuéstate y relájate en la hierba fresca y verde."
## [14] "El lago brillaba bajo el sol cálido y rojo."
## [15] "Marca el lugar con un cartel pintado de rojo."
## [16] "La cubierta del sofá y las cortinas de la sala eran azules."
## [17] "Un hombre con un suéter azul se sentó en el escritorio."
## [18] "La pequeña lámpara de neón de color rojo se apagó."
## [19] "Pinta los encajes en la pared de color verde opaco."
## [20] "Despiértate y levántate, camina hacia el verde exterior."
## [21] "La luz verde en la caja marrón parpadeaba."
## [22] "El cielo en el oeste se tiñe de rojo anaranjado."
```

• Usa el resultado anterior para identificar cuantas veces existe la coincidencia en la cadena

```
str_count(coincide, coincidencia)
```

```
## [1] 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Podemos comprobarlo haciendo:

```
str_view(coincide, coincidencia)
```

• Construye un *data frame* que en una columna tenga los enunciados que presentan coincidencia y en la siguiente el numero de coincidencias encontradas

```
data<-tibble(coincide, str_count(coincide, coincidencia))
data
```

```
## # A tibble: 22 x 2
## coincide `str_count(coincide, coincid~
## <chr> <int>
## 1 Pega la hoja en el fondo azul oscuro. 1
## 2 Instalaron azulejos verdes en la cocina. 2
```

```
## 3 Si arrojo la taza azul al suelo se romperá. 2
## 4 Dos peces azules nadaban en el tanque. 1
## 5 Una voluta de nube flotaba en el aire azul. 1
## 6 Las hojas se vuelven de color marrón y amarill~ 1
## 7 La mancha en el papel secante fue hecha por la~ 1
## 8 El cojín del sofá es de color rojo y de peso l~ 1
## 9 El cielo de la mañana era claro y azul brillan~ 1
## 10 Una grulla azul es una ave zancuda y alta. 1
## # ... with 12 more rows
```

3. Usa el conjunto de cadenas de texto de nombre oraciones para determinar cuales son las cadenas que contienen las siguientes características;

- Algún acento. Construye un subconjunto con tales cadenas

```
acento<-str_subset(oraciones, "[áéíóú]")
acento
```

```
## [1] "Las casas están construidas de ladrillos de arcilla roja."
## [2] "La caja fue arrojada al lado del camión estacionado."
## [3] "Agrega a la cuenta de la tienda hasta el último centavo."
## [4] "Si arrojo la taza azul al suelo se romperá."
## [5] "Las hojas se vuelven de color marrón y amarillo en el otoño."
## [6] "Él ordenó tarta de melocotón con helado."
## [7] "Había barro salpicado en la parte delantera de su camisa blanca."
## [8] "El cojín del sofá es de color rojo y de peso ligero."
## [9] "El médico lo curó con estas dos pastillas."
## [10] "Podían reír a pesar de que estaban tristes."
## [11] "El tercer acto era aburrido y cansó a los actores."
## [12] "El choque ocurrió cerca del banco en la calle principal."
## [13] "La lámpara brillaba con una llama verde y continua."
## [14] "El príncipe ordenó que le cortaran la cabeza."
## [15] "La planta creció grande y verde en la ventana."
## [16] "El lazo púrpura tenía diez años."
## [17] "Recuéstate y relájate en la hierba fresca y verde."
## [18] "El lago brillaba bajo el sol cálido y rojo."
## [19] "El humo salía de cada grieta."
## [20] "La cubierta del sofá y las cortinas de la sala eran azules."
## [21] "Ofreció evidencia a través de tres gráficos."
## [22] "Un hombre con un suéter azul se sentó en el escritorio."
## [23] "El sorbo de té revive a su amigo cansado."
## [24] "Una gruesa capa de pintura negra cubría todo."
## [25] "Dibuja el gráfico con líneas negras gruesas."
## [26] "La pequeña lámpara de neón de color rojo se apagó."
## [27] "Despiértate y levántate, camina hacia el verde exterior."
## [28] "La luz verde en la caja marrón parpadeaba."
## [29] "Puso su último cartucho en la pistola y disparó."
## [30] "El carnero asustó a los niños de la escuela."
## [31] "Corta una delgada lámina de la almohadilla amarilla."
## [32] "El granizo repiqueteaba en la hierba marrón quemada."
## [33] "La gran manzana roja cayó al suelo."
## [34] "El olor de la primavera hace que los corazones jóvenes salten."
```

- El número de *a,e,i,o,u* en cada cadena, para determinar cual vocal es la mas utilizada

Para ello contabilizamos el número de vocales en cada cadena y las sumamos. Esto nos da el total de veces que se utiiza la vocal en todas las cadenas. Al final construimos un data frame con los resultados

```
a<-sum(str_count(oraciones, "[á|a]"))
e<-sum(str_count(oraciones, "[é|e]"))
i<-sum(str_count(oraciones, "[í|i]"))
o<-sum(str_count(oraciones, "[ó|o]"))
u<-sum(str_count(oraciones, "[ú|u]"))
num_vocales<-tibble(a=a, e=e, i=i, o=o, u=u)
num_vocales
```

```
## # A tibble: 1 x 5
##    a      e      i      o      u
##    <int>  <int>  <int>  <int>  <int>
## 1  310    232    81     165    66
```

De esta manera identificamos que la vocal mas usada es la *a*

- Las cadenas que empiecen con una consonante y terminen con vocal

Primero seleccionamos todas las cadenas que comiencen con constante. De ese subconjunto seleccionamos las que terminen con vocal.

```
in_con_ter_voc<-str_subset(oraciones, "^[^AEIOUÁÉÍÓÚ]") %>%
str_subset("[aeiouáéíóú].$")
in_con_ter_voc
```

```
## [1] "Las casas están construidas de ladrillos de arcilla roja."
## [2] "La caja fue arrojada al lado del camión estacionado."
## [3] "Pega la hoja en el fondo azul oscuro."
## [4] "Si arrojo la taza azul al suelo se romperá."
## [5] "Dos peces azules nadaban en el tanque."
## [6] "Las hojas se vuelven de color marrón y amarillo en el otoño."
## [7] "La mancha en el papel secante fue hecha por la tinta verde."
## [8] "Había barro salpicado en la parte delantera de su camisa blanca."
## [9] "La lámpara brillaba con una llama verde y continua."
## [10] "La planta creció grande y verde en la ventana."
## [11] "Recuéstate y relájate en la hierba fresca y verde."
## [12] "Marca el lugar con un cartel pintado de rojo."
## [13] "La pequeña lámpara de neón de color rojo se apagó."
## [14] "Pinta los encajes en la pared de color verde opaco."
## [15] "La luz verde en la caja marrón parpadeaba."
## [16] "Puso su último cartucho en la pistola y disparó."
## [17] "Corta una delgada lámina de la almohadilla amarilla."
## [18] "La gran manzana roja cayó al suelo."
## [19] "Cada palabra y cada frase que habla es cierta."
```

Observa que hemos incluido un ., pues todas las oraciones terminan en punto.

Soluciones al Capítulo 9 | Manejo de fechas

Contents

Antes que nada no olvidemos las librearías necesarias

```
library(tidyverse)
library(readr)
library(lubridate)
```

1. Usa la función adecuada para convertir las siguientes cadenas de texto en fechas;

- "March 8, 2021"

```
fecha<-"March 8, 2021"
mdy(fecha)
```

```
## [1] "2021-03-08"
```

- "2021-June-07"

```
fecha<-"2021-June-07"
ymd(fecha)
```

```
## [1] "2021-06-07"
```

- “06-April-2019”

```
fecha←"06-April-2019"
dmy(fecha)
```

```
## [1] "2019-04-06"
```

- c(“August 19 (2015)”, “July 1 (2015)”)

```
fecha←c("August 19 (2015)", "July 1 (2015)")
mdy(fecha)
```

```
## [1] "2015-08-19" "2015-07-01"
```

- “12/30/20” 30 de diciembre 2020

```
fecha←"12/30/20"
mdy(fecha)
```

```
## [1] "2020-12-30"
```

2. Usa la tabla de datos de mibici y construye una gráfica donde se muestre el número de viajes por día, usando únicamente los viajes que duraron más de 20 minutos.

Cargamos la base y calculamos la duración del viaje tal y como lo habíamos hecho anteriormente

```
setwd("~/Dropbox/Curso de R/Cap9_Fechas")
library(readr)
mibici←read_csv("mibici.csv")
```

```
##
## -- Column specification --------------------------------------------------
-------
## cols(
##   viaje_id = col_double(),
##   usuario_id = col_double(),
##   genero = col_character(),
##   nacimiento = col_double(),
##   origen_id = col_double(),
##   destino_id = col_double(),
```

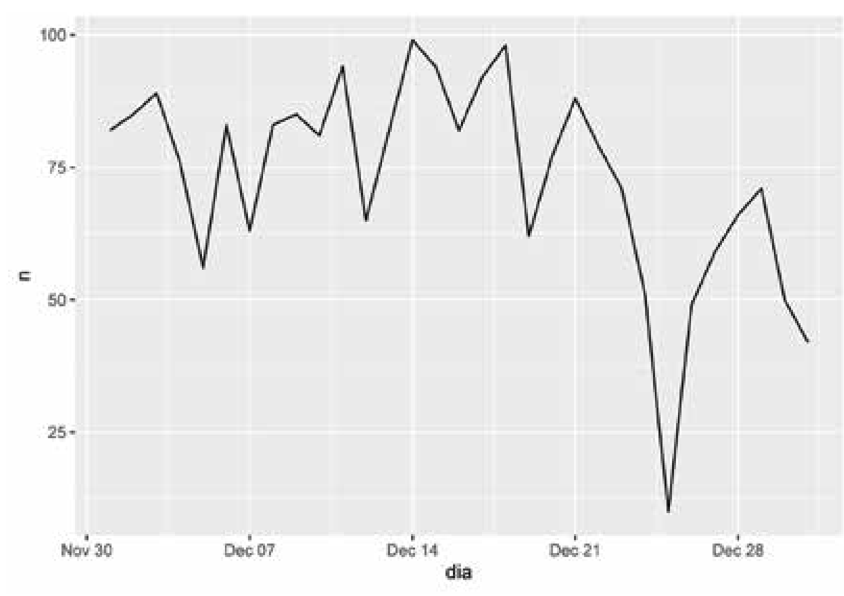


```
##   year = col_double(),
##   month = col_double(),
##   day = col_double(),
##   hour_inicio = col_double(),
##   minute_inicio = col_double(),
##   second_inicio = col_double(),
##   hour_final = col_double(),
##   minute_final = col_double(),
##   second_final = col_double()
## )
```

```
mibici2←mibici %>%
     mutate(inicio=make_datetime(year, month, day, hour_inicio, minute_ini
     cio, second_inicio))%>%
     mutate(final=make_datetime(year, month, day, hour_final, minute_final,
     second_final))%>%
     mutate(dura=final-inicio)
```

Ahora expresamos la duración en minutos, filtramos y grafícamos

```
mibici2 %>%
     mutate(dura_min=dura/dminutes(1)) %>%
     filter(dura_min≥20) %>%
     count(dia=floor_date(inicio, "day")) %>%
     ggplot(aes(dia, n)) +
     geom_line()
```



3. Escribe una función que dada una fecha de nacimiento exacta (con horas, minutos y segundos), pueda calcular la edad y expresarla como un periodo en términos de días, horas y segundos

```
nacimiento<-"12-Nov-1983 17:00:10"
dmy_hms(nacimiento)
```

```
## [1] "1983-11-12 17:00:10 UTC"
```

```
edad<-now()-dmy_hms(nacimiento)
edad
```

```
## Time difference of 13573.15 days
```

```
edad2<-as.numeric(edad)
edad2
```

```
## [1] 13573.15
```

Observa que el resultado que está en la variable **edad** contiene las palabras *Time difference of __ days* de esto solo nos interesa el valor numero expresado en días. Ahora convertimos ese valor de dias en segundos, para poder expresarlo como un periodo.

```
ddays(edad2)
```

```
## [1] "1172720381.17502s (~37.16 years)"
```

```
seconds_to_period(ddays(edad2))
```

```
## [1] "13573d 3H 39M 41.1750180721283S"
```

4. Determina que día de la semana será el 05 de junio de 2021

wday(dmy("05-06-2021"), label=TRUE)

```
## [1] Sat
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

5. Construye un periodo que tenga 5 años, 3 meses, 2 días, 15 horas, 10 minutos y 5 segundos. Luego averigua la duración del periodo en segundos

```
periodo<-years(5)+months(3)+days(2)+hours(15)+minutes(10)+seconds(5)
periodo
```

```
## [1] "5y 3m 2d 15H 10M 5S"
```

```
period_to_seconds(periodo)
```

```
## [1] 165904805
```


Soluciones al Capítulo 10 | Factores

1 Actividades

En la archivo *datos_imss.csv* se encuentra una muestra de la base de datos de trabajadores asegurados en el IMSS durante noviembre de 2020. Importa esta base de datos en un data frame llamado *imss* y calcula lo siguiente:

1. Convierte las variables *sexo, rango_edad y rango_salarial* a factores.

Cargamos la base de datos y observamos que las variables que nos interesan no son del tipo factor.

```
setwd("~/Dropbox/Curso de R/Cap10_Factores")
options(scipen=999)
library (tidyverse)
```

```
## -- Attaching packages --------------------------------------------- tidy-
verse 1.3.0 --
## v ggplot2 3.3.2 v purrr 0.3.4
## v tibble 3.0.3 v dplyr 1.0.2
## v tidyr 1.1.2 v stringr 1.4.0
## v readr 1.4.0 v forcats 0.5.0
## -- Conflicts ------------------------------------------------ tidyverse_
conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag() masks stats::lag()
```

```
library(readr)
library(forcats)
imss <- read_csv("datos_imss.csv")
```

```
##
## -- Column specification ---------------------------------------------------
-------------
## cols(
##   sexo = col_double(),
##   rango_edad = col_character(),
##   rango_salarial = col_character(),
##   ta = col_double()
## )
```

Convertimos las variables a factores

```
imss <- imss %>%
     mutate(sexo=as.factor(sexo),
          rango_edad=as.factor(rango_edad),
          rango_salarial=as.factor(rango_salarial))
```

2. Consulta el archivo *descriptor_imss* en cual encontrarás el significado de las variables *sexo, rango_edad y rango_salarial.* Utiliza esta información para renombrar los factores por otros nombres más fáciles de entender. Por ejemplo, en vez de que una etiqueta sea *E2* que significa *mayor a 1 y hasta 2 veces el salario mínimo puedes reemplazarla por 1-2 SM*.

Ahora reemplazamos los nombres de las categorías de los factores, por nombres que sean mas sencillos de entender. Para esto debemos leer el descriptor de variables, donde podemos entender a profundidad el significado de cada variable. Este casi siempre es un paso obligatorio en el manejo de tablas o bases de datos que contienen variables de tipo factor.

```
# Reemplazar factores
unique(imss$rango_salarial)
```

```
## [1] W2 W8 W4 W3 W16 W5 W7 W6 W18 W12 W14 W9 <NA> W15 W11
## [16] W10 W17 W13 W1
## Levels: W1 W10 W11 W12 W13 W14 W15 W16 W17 W18 W2 W3 W4 W5 W6 W7 W8 W9
```

```
imss <- imss %>%
mutate(rango_salarial = fct_recode(rango_salarial,
     "1 SM" = "W1",
```

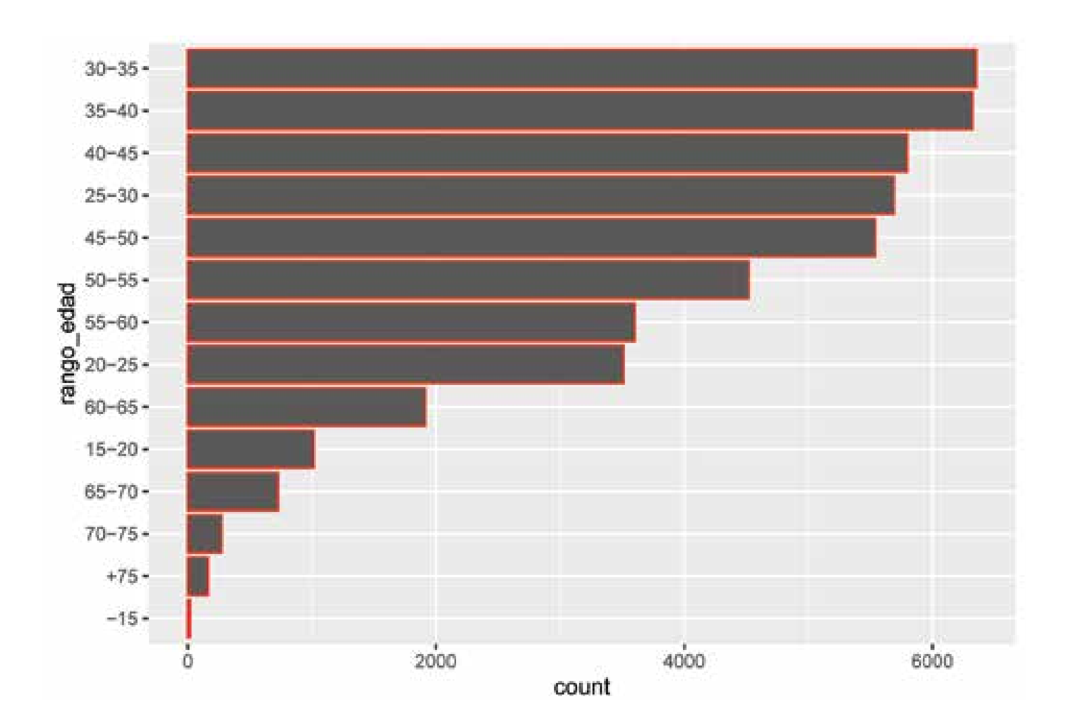


```
     "1-2 SM" = "W2",
     "2-3 SM" = "W3",
     "3-4 SM" = "W4",
     "4-5 SM" = "W5",
     "5-6 SM" = "W6",
     "6-7 SM" = "W7",
     "7-8 SM" = "W8",
     "8-9 SM" = "W9",
     "9-10 SM" = "W10",
     "10-11 SM" = "W11",
     "11-12 SM" = "W12",
     "12-13 SM" = "W13",
     "13-14 SM" = "W14",
     "14-15 SM" = "W15",
     "15-16 SM" = "W16",
     "16-17 SM" = "W17",
     "17-18 SM" = "W18"
)) %>%
mutate(sexo = fct_recode(sexo,
     "Hombre" = "1",
     "Mujer" = "2"
)) %>%
mutate(rango_edad = fct_recode(rango_edad,
     "-15" = "E1",
     "15-20" = "E2",
     "20-25" = "E3",
     "25-30" = "E4",
     "30-35" = "E5",
     "35-40" = "E6",
     "40-45" = "E7",
     "45-50" = "E8",
     "50-55" = "E9",
     "55-60" = "E10",
     "60-65" = "E11",
     "65-70" = "E12",
     "70-75" = "E13",
     "+75" = "E14"
))
```

3. Gráfica el total de trabajadores por grupo de edad

Para que la gráfica resultante sea mas interpretativa ordenamos los valores.

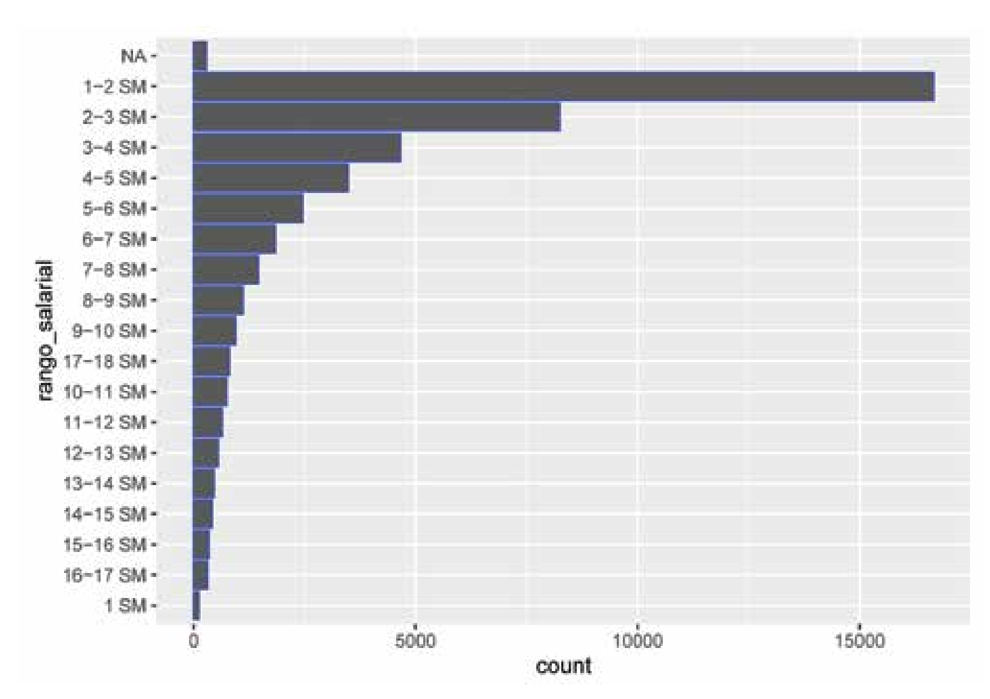
```
imss %>%
mutate(rango_edad = rango_edad %>% fct_infreq() %>% fct_rev()) %>%
ggplot(aes(rango_edad)) +
geom_bar(col="red") + coord_flip()
```



4. Gráfica el total de trabajadores por rango de salario

Nuevamente, para que la gráfica resultante sea mas interpretativa ordenamos los valores.

```
imss %>%
     mutate(rango_salarial = rango_salarial %>% fct_infreq() %>% fct_rev())
     %>%
     ggplot(aes(rango_salarial) ) +
     geom_bar(col="blue") + coord_flip()
```



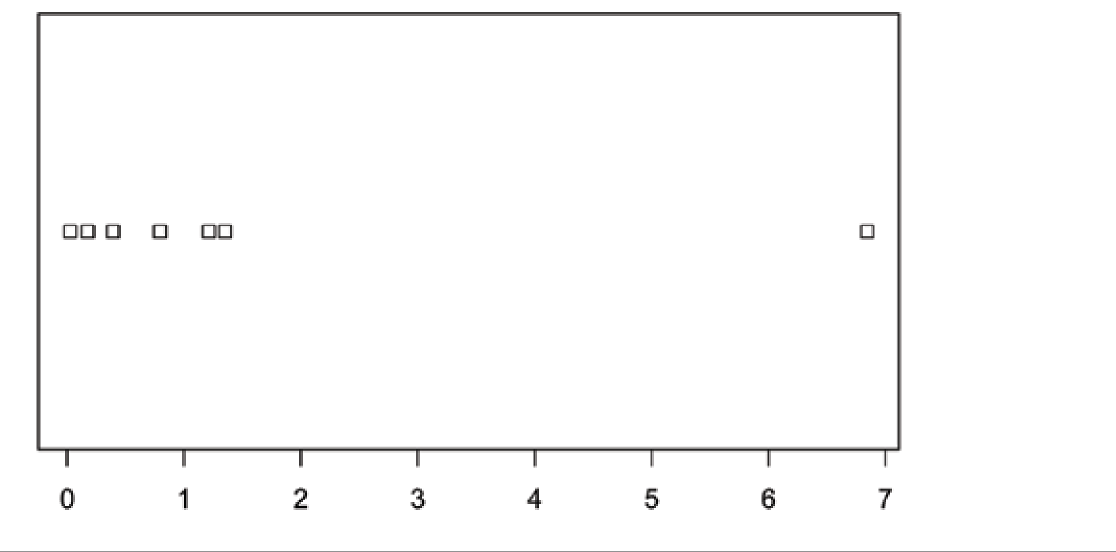
Soluciones al Capítulo 11 | Funciones

Contents

1. Modifica el siguiente código para que se obtenga exactamente el mismo resultado (una gráfica), pero con el uso de los operadores de la familia *pipe.* Para ello necesitarás la base de datos *Datos Prueba*. ¿Que ventaja observas en el uso de *pipe*?

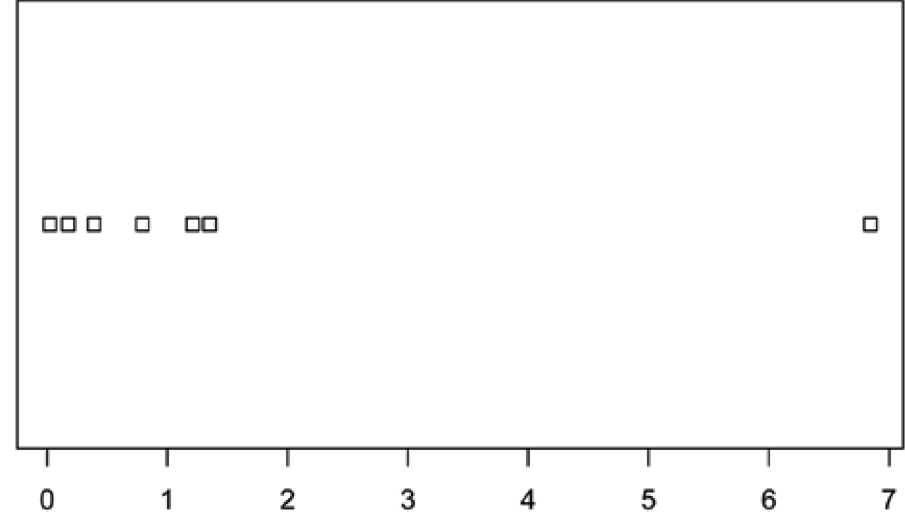
```
library(readr)
library(tidyverse)
library(magrittr)
setwd("~/Dropbox/Curso de R/Cap11_Funciones")
dp←read_csv("Datos_Prueba.csv")
```

```
dp1←select(dp, c,d)
dp2←mutate(dp1, e=c/d)
dp3←filter(dp2,e>0)
dp4←select(dp3, e)
plot(dp4)
```



El código proporcionado puede ser reemplazado por lo siguiente:

```
dp %>%
     mutate(e=c/d) %>%
     filter(e>0) %>%
     select(e) %>%
     plot()
```



La principal ventaja en este caso, es que hemos omitido la creación de bases intermedias. Debido a que el resultado final es una gráfica, no necesitamos crear una base que contenga los resultados finales. Observa también que la primera parte del código no tiene razón de ser y puede ser omitido.

2. ¿Cómo podrías hacer mas eficiente el siguiente código?

```
parte_1<-mutate(dp, h=c*d)
cor(parte_1$h, parte_1$d)
```

```
## [1] -0.4109446
```

Observa que usando %$% podemos referirnos a los nombres las variables de manera directa.

```
dp %$%
cor(c*d,d)
```

```
## [1] -0.4109446
```

3. Construye una función que pueda calcular la media ponderada de un conjunto de datos. Puedes documentarte sobre el concepto aquí Media_Ponderada. Usa el ejemplo 1 que ahí se proporciona para verificar que tu función fue construida correctamente.

Para calcular la media ponderada se requiere un conjunto de datos y los pesos relativos a cada dato. Para que funcione cada dato debe tener un peso en la estimación. Por lo que nuestra función consumirá un conjunto *x* de datos y un conjunto *w* de pesos de esos datos. Ambos deben tener la misma dimensión para que la media ponderada se pueda calcular.

```
media_ponderada<-function(x,w)
if (length(x)==length(w)){
sum((x*w)/sum(w))
} else {
print("ERROR: No existe correspondencia entre los números y los pesos")
}
```

Usamos la función

```
x<-as.numeric(c("6.4", "9.2", "8.1"))
w<-as.numeric(c(".3", ".2", ".5"))
media_ponderada(x,w)
```

```
## [1] 7.81
```

```
w2<-as.numeric(c(".3", ".2", ".5","5"))
media_ponderada(x,w2)
```

```
## [1] "ERROR: No existe correspondencia entre los números y los pesos"
```

4. Construye una función de nombre "saludo" la cual cuando se ejecute regrese, "Qué tengas un buen" y complemente con el día de la semana

En primer lugar debemos usar las funciones que nos proporciona **lubridate()** para el manejo de fechas. Del capítulo 9 recordamos que la función **wday()** nos regresa el día de una fecha y **today()** regresa la fecha actual. Con estos elementos construimos la función usando **switch()** para las diferentes opciones de día de la semana.

```
library(lubridate)
saludo<-function(dia=as.character(wday(today(), label = TRUE))) {
     switch (dia,
          Mon=print("Qué tengas un buen Lunes"),
          Tue=print("Qué tengas un buen Martes"),
          Wed=print("Qué tengas un buen Miércoles"),
          Thu=print("Qué tengas un buen Jueves"),
          Fri=print("Qué tengas un buen Viernes"),
          Sat=print("Qué tengas un buen Sábado"),
          Sun=print("Qué tengas un buen Domingo"),
     )
}
```

Observa que la función no consume ningún argumento, pues por default usa el resultado de **today()**. Observa también que no hemos incluido la opción stop ya que estamos seguros hay únicamente siete valores resultantes posibles. Ahora probamos la función

```
saludo()
```

```
## [1] "Qué tengas un buen Lunes"
```

5. Construye una función pipeable que pueda indicar si un número es par o impar.Usa el operador Módulo %%

```
par_impar<-function(x){
     if((x %% 2)==0){
          cat("El", x, " es un número PAR \n")
          invisible(x)
     } else {
          cat("El", x, " es un número IMPAR \n")
               invisible(x)
     }
}
par_impar(4)
```

```
## El 4 es un número PAR
```

```
par_impar(450311212317)
```

```
## El 450311212317 es un número IMPAR
```

Para probar que es pipeable hacemos;

```
par_impar(124545) %>%
sum(5) %>%
par_impar()
```

```
## El 124545 es un número IMPAR
## El 124550 es un número PAR
```

Observa que como resultado de la primera línea se obtiene la clasificación del número y de manera invisible se se regresa ese valor proporcionado, que se suma al 5, y nuevamente se hace la clasificación del número.

Soluciones al Capítulo 12 | Vectores

Contents

1. Construye un vector que contenga tres elementos; el primero que sea una lista con números que vayan de dos en dos hasta el 200. El segundo elemento que sea las letras a las que se puede acceder en *R* con letters. Finalmente el tercer elemento que sea el número 8. Propón un nombre cada elemento.

```
l <- list(numeros <- seq(2,200, by=2), letras <- letters, escalar <- 8)
```

Una vez que la lista sea construida:

-Accede al elemento 10 del segundo elemento de la lista construida

```
l[[2]][10]
```

```
## [1] "j"
```

-Accede al elemento 5 del tercer elemento de la lista construida

```
l[[3]][5]
```

```
## [1] NA
```

Se obtiene NA pues no existe un elemento en esa posición

-Cambia el dos contenido en la lista por 2000

```
# Antes del cambio
l[[1]][1]
```

```
## [1] 2
```

```
l[[1]][1] ← 2000
```

```
# Después del cambio
```

```
l[[1]][1]
```

```
## [1] 2000
```

2. Construye una función que pueda convertir el contenido de un vector que es del tipo *double* en tipo *entero*. Apóyate en la función *round()*.

Lo único que hacemos es solicitar que lo que contenga x, se redonde y se transforme en un número entero.

```
redondear ← function(x){
     return(as.integer(round(x,0)))
}
```

Probamos la función

```
x ← seq(1.25, 7.85, by=.2)
x
```

```
## [1] 1.25 1.45 1.65 1.85 2.05 2.25 2.45 2.65 2.85 3.05 3.25 3.45 3.65 3.85 4.05
## [16] 4.25 4.45 4.65 4.85 5.05 5.25 5.45 5.65 5.85 6.05 6.25 6.45 6.65 6.85 7.05
## [31] 7.25 7.45 7.65 7.85
```

```
typeof(x)
```

```
## [1] "double"
```

```
y ← redondear(x)
y
```



```
## [1] 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8
```

```
typeof(y)
```

```
## [1] "integer"
```

3. Crea una función para cada uno de los casos

- Consuma un vector y regrese el último valor

```
ultimo ← function(v){
v[length(v)]
}
```

Probemos con la función con dos vectores

```
a ← seq(1,23, by=3)
library(tidyverse)
a
```

```
## [1] 1 4 7 10 13 16 19 22
```

```
ultimo(a)
```

```
## [1] 22
```

```
c ←fruit[1:10]
c
```

```
## [1] "apple" "apricot" "avocado" "banana" "bell pepper"
## [6] "bilberry" "blackberry" "blackcurrant" "blood orange" "blueberry"
```

```
ultimo(c)
```

```
## [1] "blueberry"
```

- Consuma un vector y regrese el elemento en la posición 7

```
siete ← function(v){
v[7]
}
```

Probemos con la función con dos vectores

```
siete(a)
```

```
## [1] 19
```

```
siete(c)
```

```
## [1] "blackberry"
```

- Consuma un vector y regrese solo los números que sean pares

En este caso la función deberá ejecutarse únicamente sobre vectores numéricos.

```
pares <- function(v){
     if(typeof(v)=="character"){
          print("Tipo de vector inadecuado")
     } else {
          w <- (v%%2)==0
          return(v[w])
     }
}h
<- seq(1,15, by=3)
h
```

```
## [1] 1 4 7 10 13
```

```
pares(h)
```

```
## [1] 4 10
```

La función que hemos construido primero crea un vector de tipo lógico, con el nombre de w el cual regresa FALSO o VERDADERO dependiendo si el modulo de cada posición en el vector es cero. La indicación **v[w]** regresa únicamente los valores del vector original que incluyen la posición en w como verdadera.

4. Construye la siguiente lista recursiva. Una vez creada modifica lo siguiente;

```
recursiva <- list("a", "b", list("c", "d"), list("e", "f"))
```

- Cambia "a" por "z"

```
recursiva[[1]][1]
```

```
## [1] "a"
```



```
recursiva[[1]][1] <- "z"
recursiva[[1]][1]
```

```
## [1] "z"
```

- Cambia "f" por "w"

```
recursiva[[4]][2]
```

```
## [[1]]
## [1] "f"
```

```
recursiva[[4]][2] <- "w"
recursiva[[4]][2]
```

```
## [[1]]
## [1] "w"
```

- Agrega un cuarto elemento que sea un vector con los números del 5:10

```
recursiva[[5]] <- seq(5,10)
```

5. Construye una lista que tenga tres elementos. El primero de ellos que sea un data frame con tres columnas y 100 filas. El segundo elemento que sea un vector con una secuencia del 1 al 10. Finalmente el tercer elemento que sea una lista con los elementos "w", "y". Da un nombre adecuado a cada elemento e indica como sería posible acceder al contenido total de cada uno de los elementos.

Para construir la hacemos;

```
lista_gigante <- list(data_frame=tibble(
a = rnorm(100),
b = rnorm(100),
c = rnorm(100),
), vector=seq(1,10), lista=list("w", "y"))
```

Para acceder al contenido total del elemento 1, hacemos;

```
lista_gigante[["data_frame"]]
```

```
## # A tibble: 100 x 3
```

```
##    a          b          c
##    <dbl>      <dbl>      <dbl>
## 1  -0.0707    1.07       -0.00534
## 2  -0.415     0.113      -0.720
## 3  -0.814     -0.432     -0.610
## 4  0.302      -0.573     0.792
## 5  -0.742     0.343      -0.876
## 6  0.769      0.283      -1.68
## 7  0.979      -0.116     0.287
## 8  0.0818     -0.458     -1.44
## 9  0.117      -1.24      2.50
## 10 0.536      -1.67      -0.0466
## # ... with 90 more rows
```

Para el contenido del dos hacemos;

```
lista_gigante[["vector"]]
## [1] 1 2 3 4 5 6 7 8 9 10
```


Soluciones al Capítulo 13 | Iteración

Contents

Como siempre cargamos primero las librerias que necesitamos;

```
library(tidyverse)
library(purrr)
```

1. Construye conjuntos de 10 números aleatorios que con media de -5,0,5 y 50.

Usando los elementos que ya hemos aprendido.

```
medias <- c(-5,0,5,50)
map(medias, rnorm, n=10)
## [[1]]
## [1] -4.559420 -6.011424 -5.323625 -4.538867 -4.477507 -4.876357 -5.246970
## [8] -3.875120 -4.937709 -4.062863
##
## [[2]]
## [1] 0.1833448 0.6779276 1.1407944 1.2486578 -1.6338430 2.0947393
## [7] 1.1390542 1.7556139 -1.5446192 -1.8216970
##
```

```
## [[3]]
## [1] 4.256075 2.740007 5.109441 6.473592 5.231652 6.356044 4.091340 5.765539
## [9] 3.399194 3.123305
##
## [[4]]
## [1] 50.17561 51.72217 48.90076 48.30272 47.52128 50.15363 50.64729 49.99664
## [9] 51.25171 51.40706
```

O haciendo un loop

```
medias ← c(-5,0,5,50)
resultado ← "double"
for (i in medias) {
     resultado ← rnorm(10, mean = i)
     cat(resultado)
     cat("\n")
}
```

```
## -4.87796 -4.485786 -4.829358 -6.861917 -5.986511 -4.181586 -6.513429 -6.245523 -4.980027 -5.950118
## 1.078216 -0.6281853 1.378823 0.267582 0.8118165 1.8393 -0.6734541 1.107164 -1.011036 0.9555709
## 4.787356 2.836698 5.992572 3.573274 5.145304 5.942696 7.45285 5.138973 5.312097 4.616927
## 50.35303 48.33619 49.83793 50.72662 49.18137 48.64647 51.23429 49.72562 49.31959 48.44438
```

2. Haciendo babynames::births$births se obtiene el número de nacimientos registrados en Estados Unidos desde 1909 hasta 2017. Usa las funciones de la librería purrr para determinar la media, la desviación estándar, el máximo, el mínimo y la mediana de los datos

```
nacimientos ← as_tibble(babynames::births$births)
funciones ←c("mean", "sd", "max", "min", "median")
estadisticas ←map_dfr(nacimientos, ~invoke_map_dbl(funciones, x = .))
estadisticas$medida ← funciones
estadisticas
```

```
## # A tibble: 5 x 2
##    value      medida
##    <dbl>      <chr>
```



```
## 1  3478766.   mean
## 2  596912.    sd
## 3  4316233    max
## 4  2307000    min
## 5  3637000    median
```

3. Crea una función que muestre la media de cada columna de tipo numérica de un data frame

```
mostrar_media ← function(x) {
     x %>%
     keep(is.numeric) %>%
     map(mean) %>%
          str()
}
```

Probemos la función usando los datos que están cargados de manera automática en R

```
mostrar_media(iris)
```

```
## List of 4
## $ Sepal.Length: num 5.84
## $ Sepal.Width : num 3.06
## $ Petal.Length: num 3.76
## $ Petal.Width : num 1.2
```

```
mostrar_media(mtcars)
```

```
## List of 11
## $ mpg : num   20.1
## $ cyl : num   6.19
## $ disp: num   231
## $ hp :  num   147
## $ drat: num   3.6
## $ wt :  num   3.22
## $ qsec: num   17.8
## $ vs :  num   0.438
## $ am :  num   0.406
```

```
## $ gear: num  3.69
## $ carb: num  2.81
```

4. Construye una función parecida a summary, pero que opere únicamente sobre las columnas numéricas de un data frame.

```r
resumen_col <- function(x) {
x %>%
     keep(is.numeric) %>%
     map(summary)
}
```

```r
resumen_col(iris)
```

```
## $Sepal.Length
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    4.300 5.100      5.800      5.843 6.400      7.900
##
## $Sepal.Width
##    Min.  1st Qu.    Median     Mean 3rd Qu.     Max.
##    2.000 2.800      3.000      3.057 3.300      4.400
##
## $Petal.Length
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    1.000 1.600      4.350      3.758 5.100      6.900
##
## $Petal.Width
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    0.100 0.300      1.300      1.199 1.800      2.500
```

```r
resumen_col(mtcars)
```

```
## $mpg
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    10.40 15.43      19.20      20.09 22.80      33.90
##
## $cyl
##     Min. 1st Qu.    Median     Mean  3rd Qu.    Max.
```



```
##    4.000 4.000      6.000      6.188 8.000      8.000
##
## $disp
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    71.1  120.8      196.3      230.7 326.0      472.0
##
## $hp
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    52.0  96.5       123.0      146.7 180.0      335.0
##
## $drat
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    2.760 3.080      3.695      3.597 3.920      4.930
##
## $wt
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    1.513 2.581      3.325      3.217 3.610      5.424
##
## $qsec
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    14.50 16.89      17.71      17.85 18.90      22.90
##
## $vs
##    Min.   1st Qu.   Median     Mean   3rd Qu.   Max.
##    0.0000 0.0000    0.0000     0.4375 1.0000    1.0000
##
## $am
##    Min.   1st Qu.   Median     Mean   3rd Qu.   Max.
##    0.0000 0.0000    0.0000     0.4062 1.0000    1.0000
##
## $gear
##    Min.  1st Qu.    Median     Mean  3rd Qu.    Max.
##    3.000 3.000      4.000      3.688 4.000      5.000
##
```

```
## $carb
##    Min.  1st Qu.     Median      Mean  3rd Qu.     Max.
##    1.000 2.000      2.000       2.812 4.000       8.000
```

5. Haciendo uso de los elementos aprendidos, simplifica la función sum_neg_mejor que desarrollamos en este capítulo. Comprueba tu resultado

Construimos la función **suma_neg_mejor2** haciendo uso de los atajos con lo que cuenta *purrr*

```
suma_neg_mejor2 ← function(x) {
      suma_col_neg ←function(x) {
      x %>%
      as.numeric(.) %>%
      keep(~.<0) %>%
      sum(.)
      }
      sum(map_dbl(dp,suma_col_neg ))
}
```

Observa que hemos construido una función que opera dentro de otra función. La primera sólo calcula la suma de valores negativos,por columna. La segunda con la ayuda de **map** repite el proceso para todas las columnas del data frame.

Construimos unos datos de prueba para la función

```
dp ← tibble(
      a = rnorm(100),
      b = rnorm(100),
      c = rnorm(100),
)
```

Y obtenemos este resultado

```
suma_neg_mejor2(dp)
## [1] -124.0841
```

Anteriormente tuvimos que construir la función en partes. La rescribimos para comprobar que obtenemos el mismo resultado.


```
negativo←function(x){
      if(x<0) x
      else 0
}
alter←vector("double", nrow(dp))
for (i in seq(1:nrow(dp))) {
      alter[[i]]←as.numeric(negativo(dp[[i,1]]))
}
sum_neg←function(x){
      alter←vector("double", nrow(x))
      for (i in seq(1:nrow(x))) {
      alter[[i]]←as.numeric(negativo(x[[i,1]]))
      }
      return(sum(alter))
      invisible(x)
}
columna←vector("double", ncol(dp))
for (i in seq(1:ncol(dp))) {
      columna[[i]]←sum_neg(dp[i])
}
suma_neg_mejor←function(x){
      columna←vector("double", ncol(x))
      alter←vector("double", nrow(x))
      for (j in seq(1:ncol(x))) {
            for (i in seq(1:nrow(x)) ) {
            alter[[i]]←as.numeric(negativo(x[[i,j]]))
            }
            columna[[j]]←sum(alter)
      }
      return(sum(columna))
}
```

```
suma_neg_mejor(dp)
## [1] -124.0841
```

Con el uso de *purrr* hemos logrado simplificar de forma considerable nuestro código, haciendolo mas amigable y mas sencillo.

Soluciones al Capítulo 14 | Modelos Lineales

Contents

1. Retomando la base de datos de la encuesta ENIGH, realiza el mismo análisis que trabajamos en el capítulo cambiando el gasto en alimentos por gasto en salud:

-Identifica que variables están relacionadas con el gasto en salud en los hogares

Cargamos los datos y las librerías para efectuar el análisis

```
setwd("~/Dropbox/Curso de R/Cap14_Cons_Modelos")
library (tidyverse)
library (modelr)
library (readr)
library (purrr)
library (lubridate)
library (tibble)
library (moments)
library(hexbin)
enigh <- read_csv("hogares_enigh.csv")
```


Al igual que antes, definimos las variables como factores, para que sea mas entendible la interpretación de la regresión

```
# Renombrar Factores
enigh <- enigh %>%
     mutate(sexo_jefe=as.character(sexo_jefe)) %>%
     mutate(clase_hog=as.character(clase_hog)) %>%
     mutate(sexo_jefe = fct_recode(sexo_jefe, "Hombre" = "1", "Mujer" = "2")) %>%
     mutate(clase_hog = fct_recode(clase_hog,
                        "Unipersonal" = "1",
                        "Nuclear" = "2",
                        "Ampliado" = "3",
                        "Compuesto" = "4",
                        "Corresidente" = "5")) %>%
     mutate(educa_jefe = fct_recode(educa_jefe,
                        "Sin instrucción" = "01",
                        "Preescolar" = "02",
                        "Primaria incompleta" = "03",
                        "Primaria completa" = "04",
                        "Secundaria incompleta" = "05",
                        "Secundaria completa" = "06",
                        "Preparatoria incompleta" = "07",
                        "Preparatoria completa" = "08",
                        "Profesional incompleta" = "09",
                        "Presional completa" = "10",
                        "Posgrado" = "11"))
```

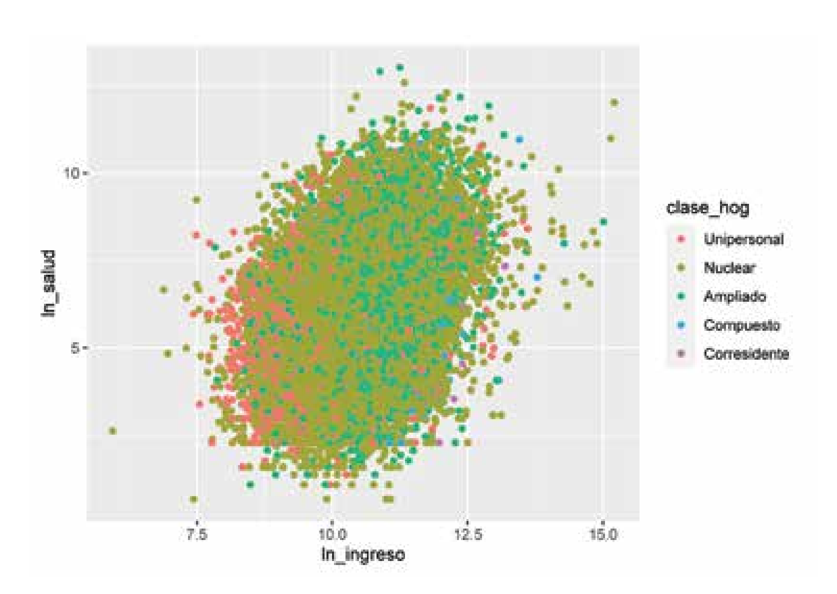
Limpiamos la base de datos, eliminando y filtrando valores perdidos. Obtenemos el logaritmo natural

```
enigh <- enigh %>%
     mutate(ing_cor = ifelse(ing_cor==0, NA, ing_cor)) %>%
     mutate(salud = ifelse(salud==0, NA, salud)) %>%
     filter(ing_cor!="NA") %>%
     filter(salud!="NA") %>%
     mutate(ln_salud=log(salud)) %>%
     mutate(ln_ingreso=log(ing_cor))
```

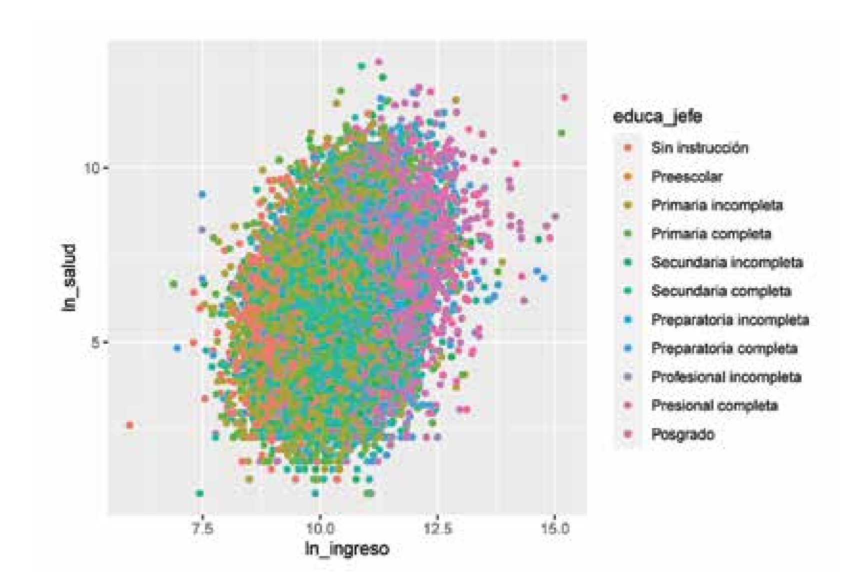
Ya que nuestra base esta lista veremos como depende el gasto en salud, según algunas de las características del hogar

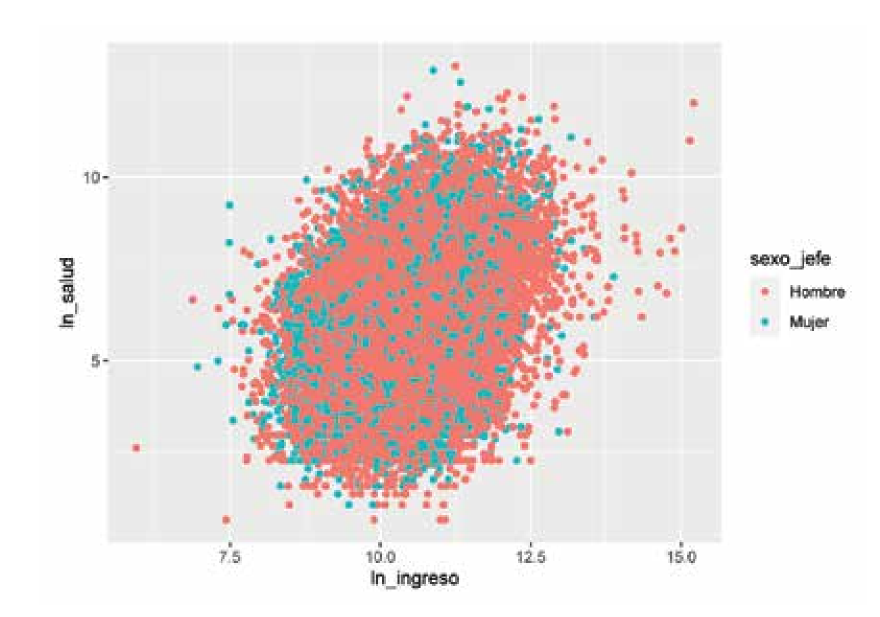
```
enigh %>%
     ggplot(aes(x = ln_ingreso, y = ln_salud)) +
```

```
     geom_point()+
     geom_point(aes(color = clase_hog))
```




```
enigh %>%
     ggplot(aes(x = ln_ingreso, y = ln_salud)) +
     geom_point()+
     geom_point(aes(color = educa_jefe))
```

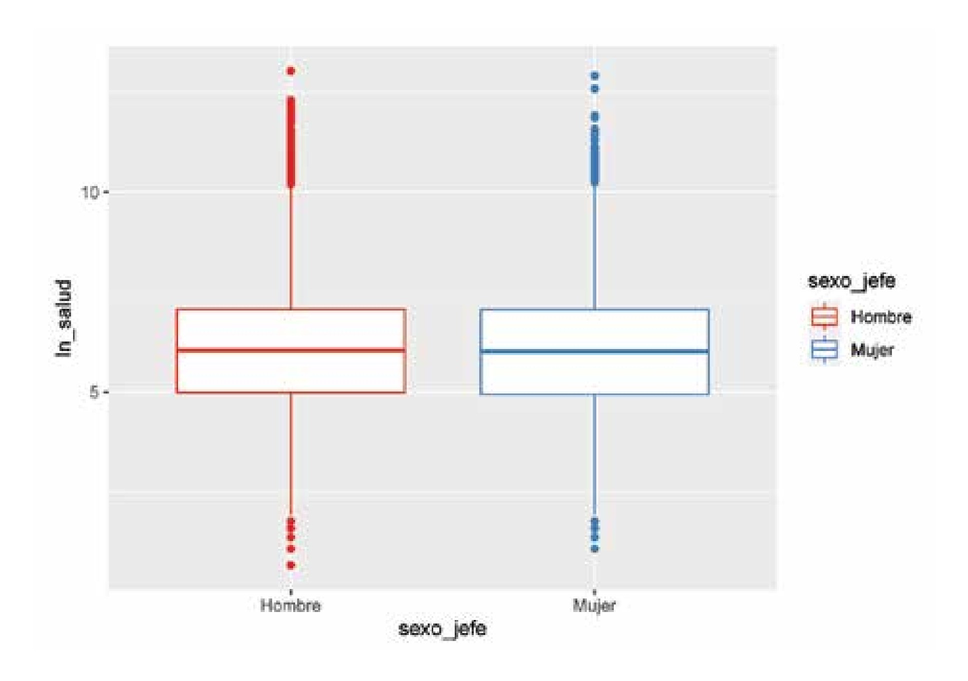





```
enigh %>%
     ggplot(aes(x = ln_ingreso, y = ln_salud)) +
     geom_point()+
     geom_point(aes(color = sexo_jefe))
```

-Efectúa una gráfica que visualmente permita contestar la siguiente pregunta ¿Existen diferencias en el gasto en salud según el sexo del jefe del hogar?

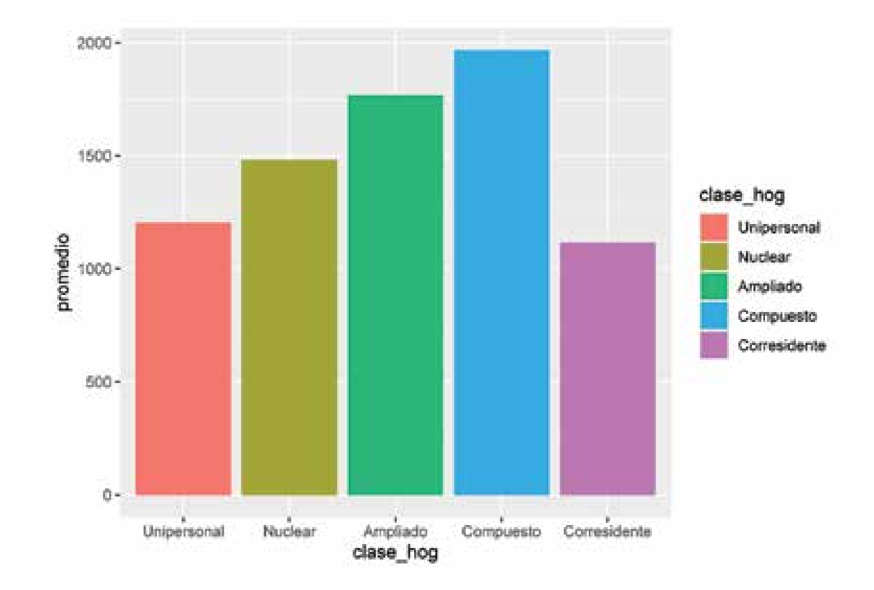
```
enigh %>%
ggplot(aes(sexo_jefe, ln_salud, color = sexo_jefe)) +
geom_boxplot() +
     scale_color_brewer(palette = "Set1")
```



El diagrama de caja nos muestra tanto los hogares con jefe de familia hombre, o jefe de familia mujer, tienen el mismo gasto en salud. Por lo que de manera visual podemos concluir que no hay diferencia entre ellos.

Calcula el gasto en salud per cápita (del hogar), diferenciando por tipo de hogar. Expresa el resultado como una gráfica

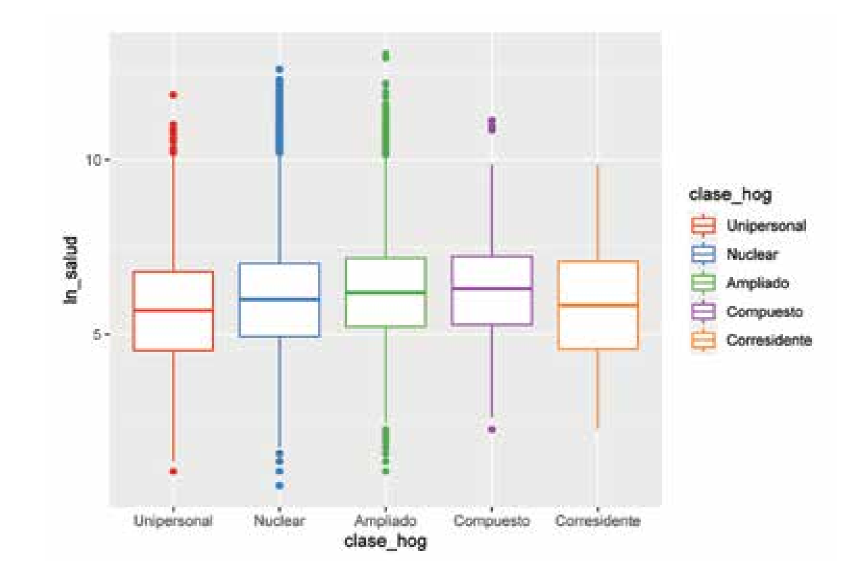
```
enigh %>%
     group_by(clase_hog) %>%
summarise(promedio=mean(salud), .groups="drop") %>%
ggplot() +
     geom_bar(aes(clase_hog,promedio, fill=clase_hog), stat = "identity")
```



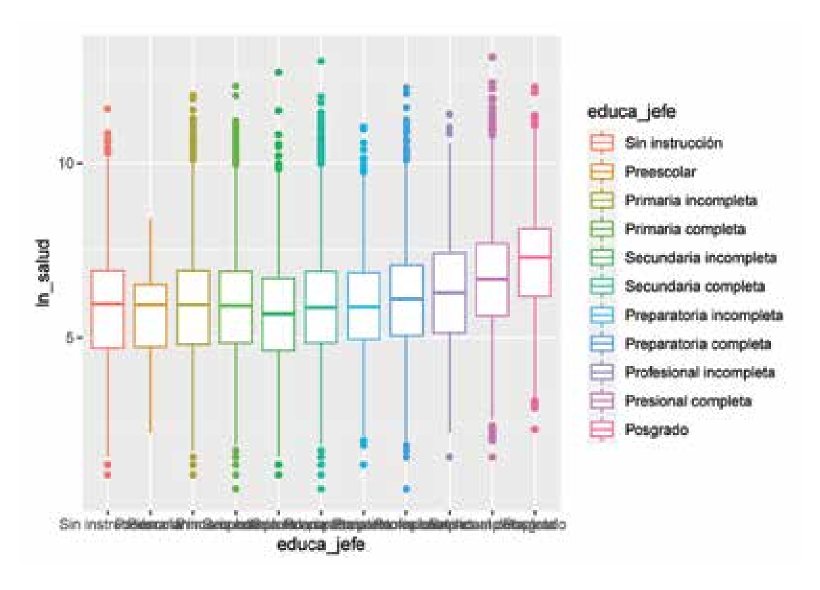
Propón un modelo de regresión lineal que explique el gasto en salud y revisa su ajuste

De los resultados anteriores parece ser que no hay diferencia entre el gasto en salud, de los hogares con jefehombre o mujer. Hagamos un diagrama de caja de las otras características del hogar para ver las diferencias

```
enigh %>%
ggplot(aes(clase_hog, ln_salud, color = clase_hog)) +
geom_boxplot() +
     scale_color_brewer(palette = "Set1")
```


```
enigh %>%
ggplot(aes(educa_jefe, ln_salud, color = educa_jefe)) +
geom_boxplot()
```



Con estos elementos generemos un modelo de regresión lineal que incluya las características del hogar previamente identificadas

```
modelo ← lm(ln_salud ~ ln_ingreso + tot_integ + edad_jefe + sexo_jefe + cla-
se_hog + educa_jefe, data=enigh)
summary(modelo, digits=5)
```

```
##
```

```
## Call:
## lm(formula = ln_salud ~ ln_ingreso + tot_integ + edad_jefe +
## sexo_jefe + clase_hog + educa_jefe, data = enigh)
##
## Residuals:
## Min 1Q Median 3Q Max
## -5.7260 -0.9780 0.0698 1.0079 6.7730
##
## Coefficients:
##                                Estimate Std. Error t value      Pr(>|t|)
## (Intercept)                     -0.6531005 0.1191250 -5.482 4.22e-08  ***
## ln_ingreso                      0.5613566  0.0119141 47.117 < 2e-16   ***
## tot_integ                       -0.0266293 0.0056415 -4.720 2.36e-06  ***
## edad_jefe                       0.0115177  0.0005945 19.375 < 2e-16   ***
## sexo_jefeMujer                  0.0101283  0.0179397 0.565  0.57237
## clase_hogNuclear                0.2112527  0.0311469 6.782  1.20e-11  ***
## clase_hogAmpliado               0.2811437  0.0377599 7.446  9.85e-14  ***
## clase_hogCompuesto              0.2219223  0.0961388 2.308  0.02098   *
## clase_hogCorresidente           -0.2986333 0.1400054 -2.133 0.03293   *
## educa_jefePreescolar            -0.0447369 0.2670349 -0.168 0.86695
## educa_jefePrimaria incompleta   -0.0040778 0.0354424 -0.115 0.90840
## educa_jefePrimaria completa     0.0032539  0.0363412 0.090  0.92866
## educa_jefeSecundaria incompleta -0.1418173 0.0528883 -2.681 0.00733   **
## educa_jefeSecundaria completa   -0.0059675  0.0363176 -0.164 0.86949
## educa_jefePreparatoria incompleta 0.0092063 0.0537010 0.171 0.86388
## educa_jefePreparatoria completa 0.1104402  0.0411120 2.686  0.00723   **
## educa_jefeProfesional incompleta 0.2148038 0.0566599 3.791  0.00015   ***
## educa_jefePresional completa    0.3664671  0.0433062 8.462  < 2e-16   ***
## educa_jefePosgrado              0.6300868  0.0650939  9.680 < 2e-16   ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.501 on 38746 degrees of freedom
## Multiple R-squared: 0.1149, Adjusted R-squared: 0.1145
```



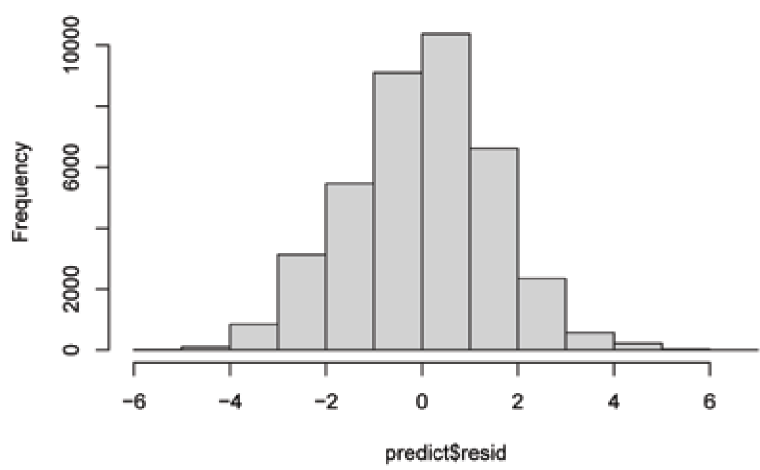
```
## F-statistic: 279.5 on 18 and 38746 DF, p-value: < 2.2e-16
```

Veamos el ajuste del modelo, primero generamos los residuos

```
predict<- enigh %>%
     add_predictions(modelo) %>%
     add_residuals(modelo)
```

```
predict %>%
     sumarise(
          media_ui = mean(resid,na.rm=TRUE),
          asimetria_ui = skewness(resid,na.rm=TRUE),
          kurtosis_ui = kurtosis(resid,na.rm=TRUE),
          cov_x1_ui = cov(ln_ingreso, resid),
          cov_x2_ui = cov(tot_integ, resid)
)
## # A tibble: 1 x 5
##    media_ui   asimetria_ui kurtosis_ui   cov_x1_ui  cov_x2_ui
##    <dbl>      <dbl>              <dbl>       <dbl>      <dbl>
## 1  -1.45e-13  -0.0536            3.02        -5.06e-15  -3.65e-15
```

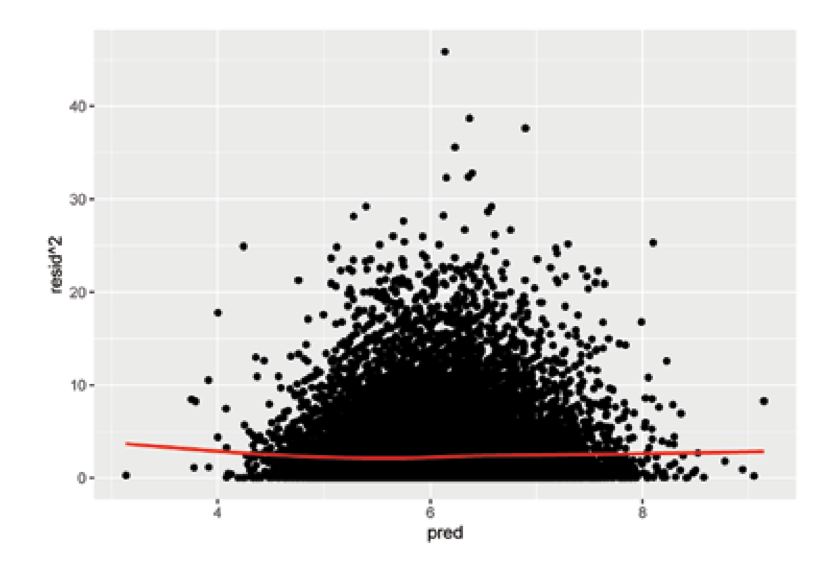
```
hist(predict$resid)
```



Vemos que la covarianza entre las variables es casi cero y que los residuos tienen una forma acampanada. Finalmente graficamos los residuos contra la variable dependiente estimada

```
predict %>%
           ggplot(aes(x = pred, y = resid^2)) +
           geom_point()+
     geom_point() +
     geom_smooth(color = "red", se = FALSE)
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



La gráfica de dispersión nos muestra que no hay relación, por lo que el ajuste es bueno.

¿Que conclusiones puedes obtener de este modelo?

Al efectuar el análisis observamos que en este caso, la variable que define el sexo del jefe del hogar no es significativa, es decir no hay diferencias en el gasto que realizan los hogares con jefe hombre o jefe mujer. Ambos gastan en promedio lo mismo. También observamos que el coeficiente estimado de

```
ln_ingreso 0.5613566 0.0119141 47.117 < 2e-16 ***
```

es significativo y positivo, lo que indica que cuando el hogar incrementa su ingreso en 1 peso, 56 centavos se dedican al gasto en salud. En el caso de la edad el coeficiente es 1

edad_jefe 0.0115177 0.0005945 19.375 < 2e-16 ***

lo que indica que por cada incremento de un año en la edad del jefe del hogar, se incrementa el gasto en salud en 1.11%.

Soluciones al Capítulo 15 | Modelos con purrr y broom

Contents

1 Realiza un modelo lineal de la esperanza de vida como función del PIB per cápita y calcula lo siguiente:

-El modelo de regresión para cada país. Efectúa un resumen del país 22

Cargamos nuestros datos como es costumbre y damos de alta las librerías que necesitaremos.

```
setwd("~/Dropbox/Curso de R/Cap15_Varios_Modelos")
library (tidyverse)
library (modelr)
library (readr)
library (purrr)
library(broom)
data <- read_csv("indicadores_banco_mudial.csv")
colnames(data)
```

```
## [1] "país Name"
## [2] "país Code"
## [3] "tiempo"
## [4] "tiempo Code"
## [5] "Esperanza de vida al nacer, total (años) [SP.DYN.LE00.IN]"
## [6] "PIB per cápita (US$ a precios constantes de 2010) [NY.GDP.PCAP.KD]"
```

Igual que antes, renombramos algunas variables para un manejo mas sencillo

```
names(data)[2] <- "pais"
names(data)[5] <- "esperanza"
names(data)[6] <- "pib_pc"
```

Anidamos los datos y generamos todos los modelos

```
grupo_paises <- data %>%
group_by(pais) %>%
nest()


modelo_pais <- function(df) {
lm(esperanza ~ pib_pc, data = df)
}
reg_pais <- map(grupo_paises$data, modelo_pais)
```

Para efectuar el resumen, debemos recordarlo lo que hemos aprendido de vectores.

```
summary(reg_pais[[22]])
```

```
##
## Call:
## lm(formula = esperanza ~ pib_pc, data = df)
##
## Residuals:
##    Min        1Q    Median 3Q Max
##    -4.0099    -1.4742 -0.2137  1.4907 2.8202
##
## Coefficients:
##               Estimate   Std.  Error                        t value Pr(>|t|)
## (Intercept)   5.406e+01        1.093e+00 49.45 <2e-16       ***
## pib_pc        1.967e-03        1.688e-04 11.65 <2e-16       ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.801 on 57 degrees of freedom
## Multiple R-squared: 0.7043, Adjusted R-squared: 0.6991
## F-statistic: 135.7 on 1 and 57 DF, p-value: < 2.2e-16
```


Esto nos muestra los elementos mas importantes del modelo estimado específicamente para el país número 22. Si queremos saber que país es, debemos identificar su ordenamiento en el objeto anidado **grupo_paises** y **data**

```
esp <- data %>%
     filter(pais==grupo_paises[[1]][22])
unique(esp$`pais Name`)
```

```
## [1] "Suriname"
```

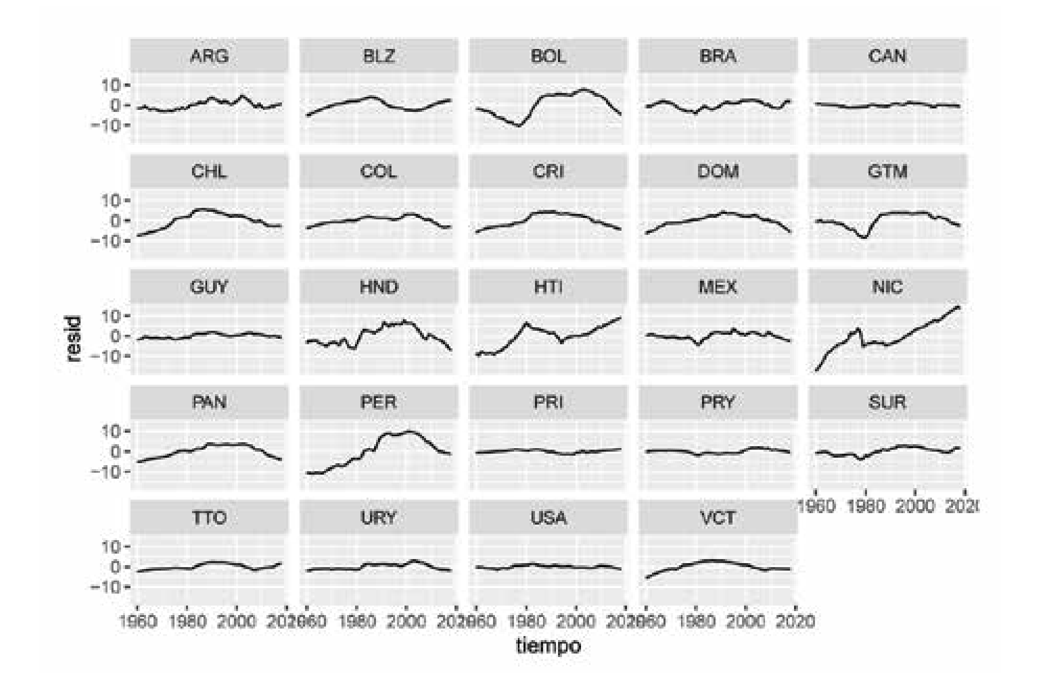
-Analiza el ajuste de los residuos de cada país. Di en cuales de ellos es poco probable que exista una relación lineal entre la esperanza de vida y el producto interno bruto

Generamos los errores para cada uno de los modelos y los graficamos

```
grupo_paises <- grupo_paises %>%
mutate(modelo = map(data, modelo_pais)) %>%
     mutate(resids = map2(data, modelo, add_residuals))

residuos <- unnest(grupo_paises, resids)
```

```
residuos %>%
     ggplot(aes(tiempo, resid)) +
     geom_line() +
     facet_wrap(~pais)
```



A simple vista podemos observar que en los siguiente países, los errores que genera una modelo de regresión lineal, son mas altos que el resto de los países. *BOL, CHL, GTM, HND, HTI, NIC, ER*

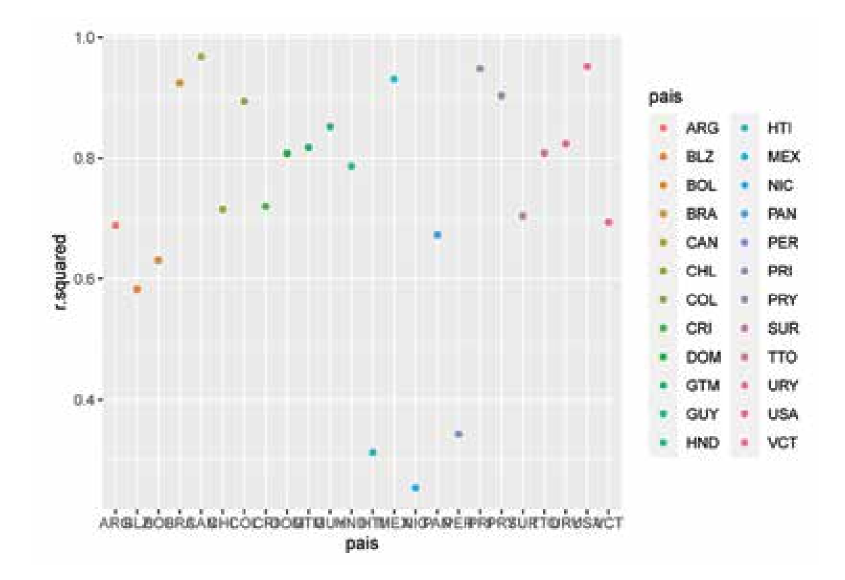
Calcula las métricas de ajuste utilizando glance

En este caso tenemos el r-cuadrado para cada el modelo de cada uno de los países

```
ajuste ← grupo_paises %>%
mutate(ajustes = map(modelo, broom::glance)) %>%
unnest(ajustes)
```

Y mostramos el ajuste.

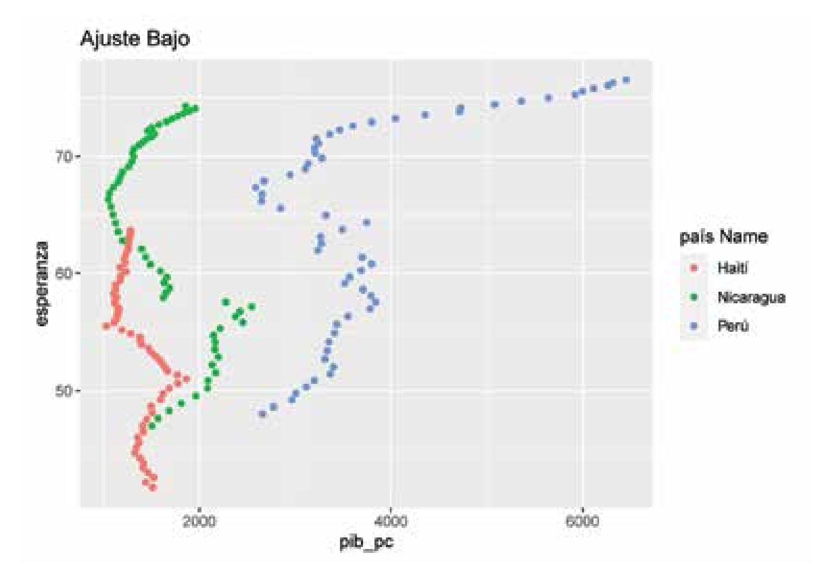
```
ajuste %>%
ggplot(aes(pais, r.squared)) +
geom_point(aes(color = pais))
```



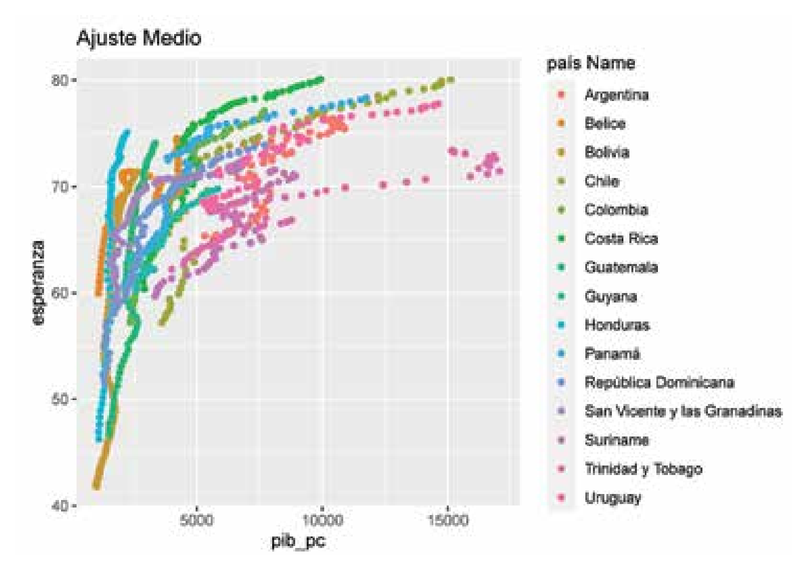
-Efectúa tres gráfica. En una pon lo modelos que tiene un ajuste menor o igual a .5, en otra los que tienen un ajuste entre .5 y .9. En la tercera incluye aquellos que tienen ajuste mayor a .9 En todos los casos gráfica la relación entre el PIB y la esperanza de vida. Usa la función ggtitle para agregar el titulo del gráfico

En el objeto **ajuste** se encuentra en *r-cuadrado* asociado a cada país. Por lo que debemos comenzar haciendo un filtro. Debido a que los datos del PIB y de la esperanza de vida, se encuentra en el objeto data es necesario efectuar un joint para poder juntar todos los datos y así efectuar la gráfica. En cada caso filtramos según se requiere.


```
data %>%
     semi_join(filter(ajuste, adj.r.squared ≤0.5), by = "pais") %>%
     ggplot(aes(pib_pc, esperanza, color =`país Name`)) +
     geom_point()+
     ggtitle("Ajuste Bajo")
```

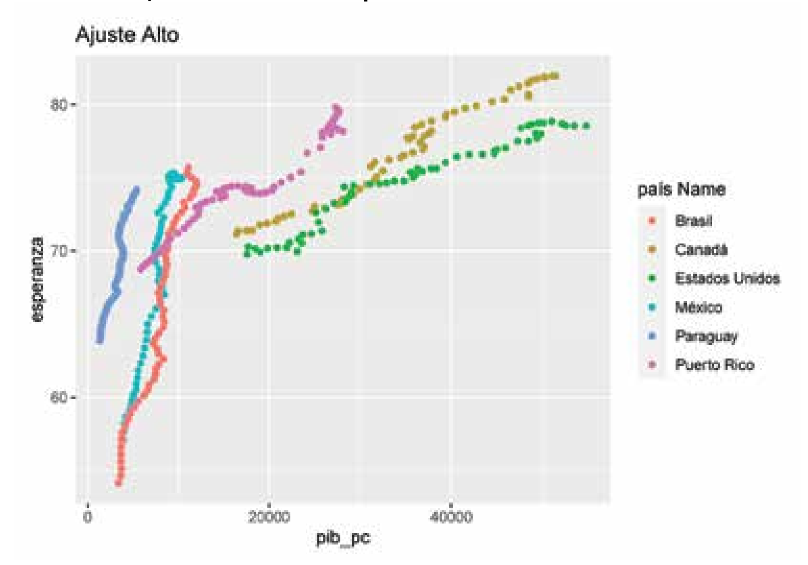



```
data %>%
     semi_join(filter(ajuste, adj.r.squared>0.5 & adj.r.squared≤0.9), by =
     "pais") %>%
     ggplot(aes(pib_pc, esperanza, color =`país Name`)) +
     geom_point()+
     ggtitle("Ajuste Medio")
```



```
data %>%
     semi_join(filter(ajuste, adj.r.squared>0.9), by = "pais") %>%
     ggplot(aes(pib_pc, esperanza, color =`país Name`)) +
     geom_point()+
     ggtitle("Ajuste Alto")
```

-Para este conjunto de datos, determina en promedio como ha cambiado la esperanza de vida



de 1960 a 2018

```
data %>%
     group_by(tiempo) %>%
     summarise(Prom_EV=mean(esperanza))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## # A tibble: 59 x 2
##    tiempo     Prom_EV
##    <dbl>      <dbl>
## 1  1960       57.4
## 2  1961       57.9
## 3  1962       58.3
## 4  1963       58.6
## 5  1964       59.0
## 6  1965       59.4
## 7  1966       59.8
## 8  1967       60.2
## 9  1968       60.6
## 10 1969       61.0
## # ... with 49 more rows
```

Al efectuar el promedio por año, tenemos que en 1960, la esperanza de vida era de 57.41 años, mientras que en 2018, es de 75.22. Por lo cual ha aumentado casi 20 años.

Soluciones al Capítulo 16 | RMarkdown

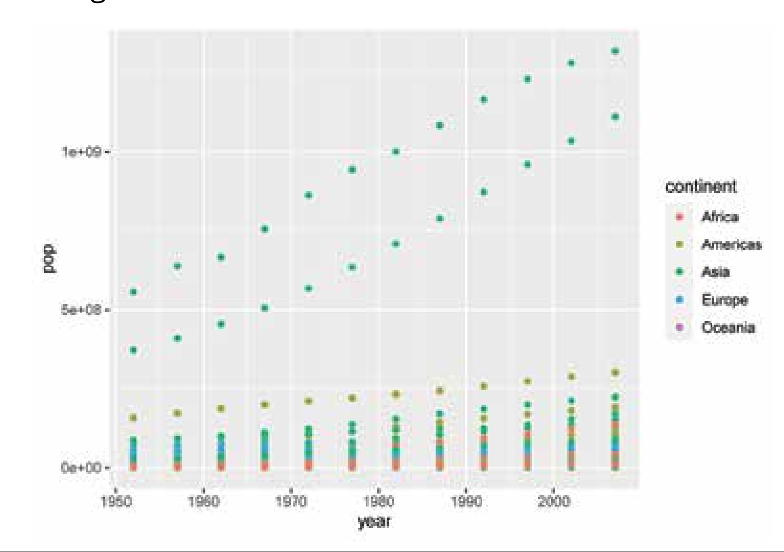
Contents

1. Crea un nuevo archivo tipo *RMarkdown*. Insertar un *chunk* y sobre el, escribe el siguiente código. Modifica los encabezados del *chunk* para que sólo sea posible observar el resultado de la ejecución

Será primero necesario iniciar un documento *Rmarkdown.* Una vez creado debemos asegurarno que contiene los siguientes elementos:

```
```{r, echo=FALSE, message=FALSE}
library(tidyverse)
library(gapminder)
gapminder %>%
     ggplot()+
     geom_point(aes(year, pop, col=continent))
```
```

Esto mostrara la siguiente gráfica en el documento final



2. Crear la siguiente estructura en un archivo RMarKdown.

En este caso sólo debemos asegurarnos que en la salida del documento hemos incluido la opción toc para la tabla de contenidos y la opción **number_sections** para la numeración.

```
---
title: 'El Ingreso Gasto de los Hogares'
output:
     html_document:
          toc: yes
          number_sections: yes
---

# Introducción
## Conociendo la base de datos
## Especificaciones

# El principal Ingreso
## Salarios
## Negocios

# El principal gasto
## Comida
## Educación
```

3. Usa la librería gapminder para obtener el conjunto de datos contenidos en ella y generar el siguiente dashboard

Debemos crear un archivo *RMarkdown* y asegurarnos que contenga la siguiente información Recuerda que en este tipo de documentos

```
# Define una pestaña
## Define una columna
### Define una fila
```

```
---
title: "Dashboard"
output: flexdashboard::flex_dashboard
---

```{r setup, include=FALSE}
library(flexdashboard)
```



```
library(gapminder)
library(tidyverse)
library(DT)
```

GAPMINDER

Columna 1

Población

```{r}
gapminder %>%
ggplot()+
geom_point(aes(year, pop, col=continent))
```

Producto Interno Bruto

```{r}
gapminder %>%
arrange(desc(gdpPercap)) %>%
select(country, year, gdpPercap) %>%
DT::datatable()
```

Columna 2

Expectativa de vida

```{r}
gapminder %>%
ggplot()+
geom_point(aes(lifeExp, pop, col=continent))
```
```

Soluciones al Capítulo 17 | Redux con ggplot2

Contents

1. La base de datos airquality contiene los datos registrados de la calidad del aire en Nueva York del mes de Mayo a Septiembre de 1973. Esta base forma parte de los data set de ejemplo de R. Úsala para construir la siguiente gráfica. Para este ejercicio deberás revisar la documentación de theme y hacer uso de varios de los elementos que aprendimos en capítulos anteriores. Para los cortes del eje horizontal usa la función scale_x_date(breaks="2weeks")

Primero cargamos los librerías que necesitaremos así como los datos, y observamos que necesitamos construir una variable de fecha, ya que no se incluye. Sabemos que los datos son del año 1973, así que podemos hacer lo siguiente.

```
library(lubridate)
library(datasets)
library(lubridate)
library(tidyverse)
library(ggrepel)
datos <- airquality %>%
     mutate(year=1973)%>%
     mutate(fecha=make_date(year, Month, Day))
head(datos)
```

```
##    Ozone      Solar.R     Wind Temp Month Day    year  fecha
## 1  41         190         7.4   67    5     1     1973 1973-05-01
## 2  36         118         8.0   72    5     2     1973 1973-05-02
## 3  12         149         12.6  74    5     3     1973 1973-05-03
## 4  18         313         11.5  62    5     4     1973 1973-05-04
## 5  NA         NA          14.3  56    5     5     1973 1973-05-05
## 6  28         NA          14.9  66    5     6     1973 1973-05-06
```


En la gráfica se observa el punto donde se registro el nivel máximo de temperatura, por lo que sera necesarioidentificarlo. Para ello creamos el siguiente objeto.

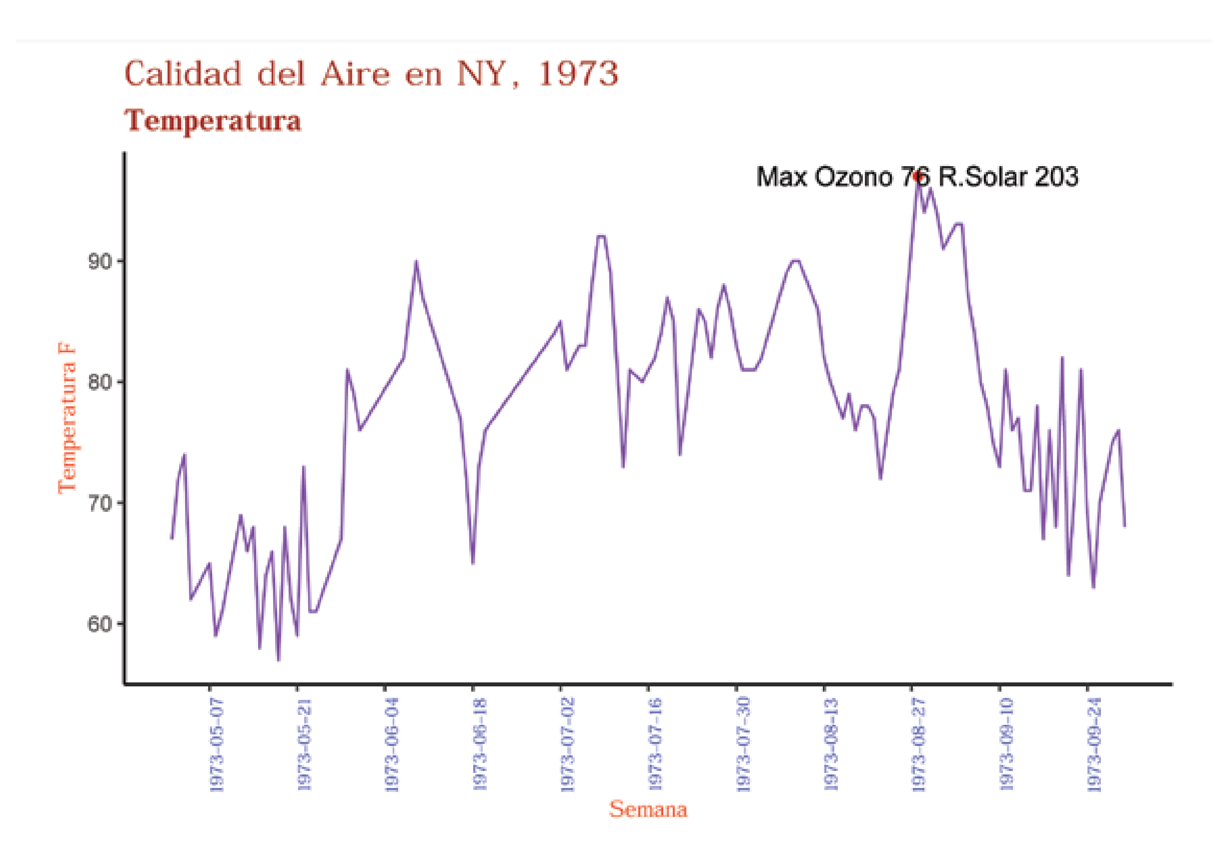
```
maximo <- filter(datos, Temp==max(datos$Temp))
```

Ya con los datos procedemos a efectuar la gráfica. Primero definamos los elementos que modificaremos en la gráfica.

```
titulo <- element_text(family = "Comic Sans MS", face = "bold", color="-
brown", size=14)
subtitulo <- element_text(family = "Comic Sans MS", face = "italic", color="-
brown", size=12)
texto_eje<- element_text(family = "Comic Sans MS", face = "plain", c
lor="red", size=9)
etiq_x <- element_text(family = "Comic Sans MS", face = "plain", color="-
blue", size=7, angle = 90)
```

Lo que sigue será la elaborar la gráfica. En este caso tendremos tres geometrías, una de linea, una de puntos, con la que identificaremos el punto de máxima temperatura y una tercer con la cual incluiremos el texto deseado, sobre ese punto.

```
ggplot(na.omit(datos))+
     geom_line(aes(fecha, Temp),col="purple")+theme_classic()+
     labs(title="Calidad del Aire en NY, 1973",
          subtitle="Temperatura",
          x="Semana",
          y="Temperatura F")+
     theme(plot.title = titulo,
          plot.subtitle =subtitulo,
          axis.title.x = texto_eje,
          axis.title.y = texto_eje,
          axis.text.x =etiq_x) +
     scale_x_date(breaks="2 weeks") +
     geom_point(aes(fecha, Temp), data=maximo, size=1.5, col="red")+
     geom_text(aes(fecha, Temp, label=(paste("Max","Ozono", Ozone,
     "R.Solar",   Solar.R))) ,data=maximo)
```



En esta gráfica hemos modificado los colores de las marcas del eje *x*. Para ello usamos dentro del **theme** usamos la función **axis.text.x** la cual tomará los parámetros establecidos en **etiq_x <- element_text(family = "Comic Sans MS", face = "plain", color="blue", size=7, angle = 90)** aquí incluimos la opción angle = 90 para especificar la inclinación de la etiqueta.

2. Usa la base de datos gapminderque contiene información sobre diferentes variable macroeconómicas de varios países del mundo. En estos países se encuentra México. Usa la información para replicar la siguiente gráfica. La población está expresada en términos logarítmicos log(pop) y las etiquetes del eje "x" las obtuvimos usando scale_x_log10(labels = scales::dollar). Los puntos que sobresalen al resto corresponde a la evolución de México.

Observamos que en este caso necesitaremos separar los datos de México, ya que son los que se encuentran sobrepuestos al resto de puntos. Para ello hacemos lo siguiente;

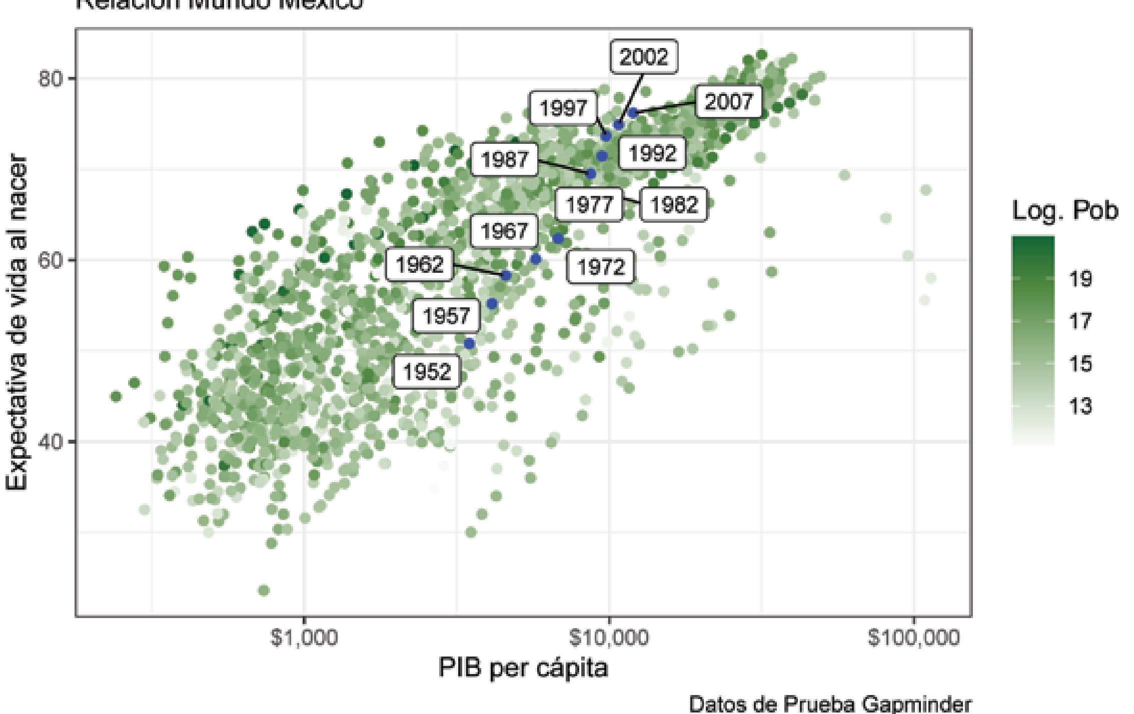
```
Mexico ← gapminder %>%
     filter(country=="Mexico") %>%
     mutate(log_pop = log(pop))
```

La indicación nos advierte que la población se ha transformado en términos logarítmicos.

Para elaborar la gráfica necesitaremos tres geometrías; una para todos los puntos, una para los puntos exclusivos de México y una más, para agregar las etiquetas sobre los datos de México.

El resto de los elementos de la gráfica se pueden modificar con las funciones que se trabajaron en el capítulo.

```
g1← gapminder %>%
     mutate(log_pop = log(pop)) %>%
     ggplot(aes(x = gdpPercap, y = lifeExp, color = log_pop)) + theme_bw()+
     geom_point()+
     scale_x_log10(labels = scales::dollar) +
     labs(x = "PIB per cápita", y = "Expectativa de vida al nacer",
          color = "Log. Pob", title = "Relación Exp. Vida y PIB",
          subtitle="Relación Mundo México",
          caption="Datos de Prueba Gapminder")
g1+geom_point(aes(x = gdpPercap, y = lifeExp), data=Mexico, col="blue")+
     geom_label_repel(aes(label = year), data = Mexico, size=3, col="black")+
     scale_colour_gradient(low="white", high="darkgreen")
```





A2 CAPACITACIÓN

